Stackoverflow about Python

hackerxu

Published with GitBook

Table of Contents

book passing

打发无聊时间翻译的Stack Overflow关于Python的部分,一来是为了学习Python,二来是为了学习英语,就这样.

PS:按vote排名翻译

待翻译

13 http://effbot.org/zone/default-values.htm

20 http://stackoverflow.com/questions/1436703/difference-between-str-and-repr-in-python

35 http://stackoverflow.com/questions/3684484/peak-detection-in-a-2d-array

64 http://stackoverflow.com/questions/1301346/the-meaning-of-a-single-and-a-double-underscore-before-an-object-name-in-python 部分6

rank	▲	☆	vote	url
1	2888	2315	4033	url

Python中关键字yield有什么作用?

yield有什么用?

例如下面这段代码:

```
def node._get_child_candidates(self, distance, min_dist, max_dist):
    if self._leftchild and distance - max_dist < self._median:
        yield self._leftchild
    if self._rightchild and distance + max_dist >= self._median:
        yield self._rightchild
```

下面是调用它:

```
result, candidates = list(), [self]
while candidates:
    node = candidates.pop()
    distance = node._get_dist(obj)
    if distance <= max_dist and distance >= min_dist:
        result.extend(node._values)
    candidates.extend(node._get_child_candidates(distance, min_dist, max_dist))
return result
```

当 `_get_child_candidates` 方法被调用的时候发生了什么?是返回一个列表?还是一个元祖?它还能第二次调用吗?后面的调用什么时候结束?

为了理解yield有什么用,首先得理解generators,而理解generators前还要理解iterables

Iterables

当你创建了一个列表,你可以一个一个的读取它的每一项,这叫做iteration:

```
>>> mylist = [1, 2, 3]
>>> for i in mylist:
...    print(i)
1
2
3
```

Mylist是可迭代的.当你用列表推导式的时候,你就创建了一个列表,而这个列表也是可迭代的:

```
>>> mylist = [x*x for x in range(3)]
>>> for i in mylist:
...     print(i)
0
1
4
```

所有你可以用在 `for...in...` 语句中的都是可迭代的:比如lists,strings,files...因为这些可迭代的对象你可以随意的读取所以非常方便易用,但是你必须把它们的值放到内存里,当它们有很多值时就会消耗太多的内存.

Generators

生成器也是迭代器的一种,但是你只能迭代它们一次.原因很简单,因为它们不是全部存在内存里,它们只在要调用的时候在内存里生成:

```
>>> mygenerator = (x*x for x in range(3))
>>> for i in mygenerator:
...     print(i)
0
1
4
```

生成器和迭代器的区别就是用 `()` 代替 `[]` ,还有你不能用 `for i in mygenerator` 第二次调用生成器:首先计算0,然后会在内存里丢掉0去计算1,直到计算完4.

Yield

`Yield` 的用法和关键字 `return` 差不多,下面的函数将会返回一个生成器:

```
>>> def createGenerator():
...     mylist = range(3)
...     for i in mylist:
...         yield i*i
...
>>> mygenerator = createGenerator() # 创建生成器
>>> print(mygenerator) # mygenerator is an object!
<generator object createGenerator at 0xb7555c34>
>>> for i in mygenerator:
...      print(i)
0
1
4
```

在这里这个例子好像没什么用,不过当你的函数要返回一个非常大的集合并且你希望只读一次的话,那么它就非常的方便了.

要理解 `Yield` 你必须先理解当你调用函数的时候,函数里的代码并没有运行.函数仅仅返回生成器对象,这就是它最微妙的地方:-)

然后呢,每当 `for` 语句迭代生成器的时候你的代码才会运转.

现在,到了最难的部分:

当 `for` 语句第一次调用函数里返回的生成器对象,函数里的代码就开始运作,直到碰到 `yield` ,然后会返回本次循环的第一个返回值.所以下一次调用也将运行一次循环然后返回下一个值,直到没有值可以返回.

一旦函数运行并没有碰到 `yeild` 语句就认为生成器已经为空了.原因有可能是循环结束或者没有满足 `if/else` 之类的.

对于你的代码的解释

生成器:

```
# 这里你创建node方法的对象将会返回一个生成器
def node._get_child_candidates(self, distance, min_dist, max_dist):

  # 这里的代码你每次使用生成器对象的时候将会调用

  if self._leftchild and distance - max_dist < self._median:
      yield self._leftchild

  if self._rightchild and distance + max_dist >= self._median:
      yield self._rightchild

  # 如果代码运行到这里,生成器就被认为变成了空的
```

调用:

```
# 创建空列表和一个当前对象索引的列表
result, candidates = list(), [self]

# 在candidates上进行循环(在开始只保含一个元素)
while candidates:

    # 获得最后一个condidate然后从列表里删除
    node = candidates.pop()

    # 获取obj和candidate的distance
    distance = node._get_dist(obj)

    # 如果distance何时将会填入result
    if distance <= max_dist and distance >= min_dist:
        result.extend(node._values)

    candidates.extend(node._get_child_candidates(distance, min_dist, max_dist))

return result
```

这段代码有几个有意思的地方:

- 一般的时候我们会在循环迭代一个列表的同时在列表中添加元素:-)尽管在有限循环里结束多少有一些危险,但也不失为一个简单的方法去遍历嵌套的数据.在这里 `candidates.extend(node._get_child_candidates(distance, min_dist, max_dist))` 将遍历生成器的每一个值,但是 `while` 循环中的condidates将不再保存已经遍历过的生成器对象，也就是说添加进condidates的生成器对象只会遍历一遍。
- `extend()` 是一个列表对象的方法,它可以把一个迭代对象添加进列表.

我们经常这么用:

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a.extend(b)
>>> print(a)
[1, 2, 3, 4]
```

但是在你给的代码里得到的是生成器,这样做的好处:

1. 你不需要读这个值两次
2. 你能得到许多孩子节点但是你不希望他们全部存入内存.

这种方法之所以能很好的运行是因为Python不关心方法的参数是不是一个列表.它只希望接受一个迭代器,所以不管是strings,lists,tuples或者generators都可以!这种方法叫做duck typing,这也是Python看起来特别cool的原因之一.但是这又是另外一个传说了,另一个问题~~

好了,看到这里可以打住了,下面让我们看看生成器的高级用法:

控制迭代器的穷尽

```
>>> class Bank(): # 让我们建个银行,生产许多ATM
...    crisis = False
...    def create_atm(self):
...        while not self.crisis:
...            yield "$100"
>>> hsbc = Bank() # 当一切就绪了你想要多少ATM就给你多少
>>> corner_street_atm = hsbc.create_atm()
>>> print(corner_street_atm.next())
$100
>>> print(corner_street_atm.next())
$100
>>> print([corner_street_atm.next() for cash in range(5)])
['$100', '$100', '$100', '$100', '$100']
>>> hsbc.crisis = True # cao,经济危机来了没有钱了!
>>> print(corner_street_atm.next())
<type 'exceptions.StopIteration'>
>>> wall_street_atm = hsbc.create_atm() # 对于其他ATM,它还是True
>>> print(wall_street_atm.next())
<type 'exceptions.StopIteration'>
>>> hsbc.crisis = False # 麻烦的是,尽管危机过去了,ATM还是空的
>>> print(corner_street_atm.next())
<type 'exceptions.StopIteration'>
>>> brand_new_atm = hsbc.create_atm() # 只能重新新建一个bank了
>>> for cash in brand_new_atm:
...    print cash
$100
$100
$100
$100
$100
$100
$100
$100
$100
...
```

它对于一些不断变化的值很有用,像控制你资源的访问.

Itertools,你的好基友

itertools模块包含了一些特殊的函数可以操作可迭代对象.有没有想过复制一个生成器?链接两个生成器?把嵌套列表里的值组织成一个列表?Map/Zip还不用创建另一个列表?

来吧 `import itertools`

来一个例子?让我们看看4匹马比赛有多少个排名结果:

```
>>> horses = [1, 2, 3, 4]
>>> races = itertools.permutations(horses)
>>> print(races)
<itertools.permutations object at 0xb754f1dc>
>>> print(list(itertools.permutations(horses)))
[(1, 2, 3, 4),
 (1, 2, 4, 3),
 (1, 3, 2, 4),
 (1, 3, 4, 2),
 (1, 4, 2, 3),
 (1, 4, 3, 2),
 (2, 1, 3, 4),
 (2, 1, 4, 3),
 (2, 3, 1, 4),
 (2, 3, 4, 1),
 (2, 4, 1, 3),
 (2, 4, 3, 1),
 (3, 1, 2, 4),
 (3, 1, 4, 2),
 (3, 2, 1, 4),
 (3, 2, 4, 1),
 (3, 4, 1, 2),
 (3, 4, 2, 1),
 (4, 1, 2, 3),
 (4, 1, 3, 2),
 (4, 2, 1, 3),
 (4, 2, 3, 1),
 (4, 3, 1, 2),
 (4, 3, 2, 1)]
```

理解迭代的内部机制

迭代是可迭代对象(对应 `__iter__()` 方法)和迭代器(对应 `__next__()` 方法)的一个过程.可迭代对象就是任何你可以迭代的对象(废话啊).迭代器就是可以让你迭代可迭代对象的对象(有点绕口,意思就是这个意思)

预知后事如何,请看[for 循环是如何工作的]

rank	▲	✰	vote	url
2	1919	1842	3137	url

Python中的元类(metaclass)是什么?

元类是什么?如何使用元类?

类对象

在理解元类之前,你需要掌握Python里的类.Python中类的概念借鉴于Smalltalk，这显得有些奇特.

在大多数语言中,类就是一组用来描述如何生成一个对象的代码段。在Python中这一点仍然成立：

```
>>> class ObjectCreator(object):
...       pass
...

>>> my_object = ObjectCreator()
>>> print(my_object)
<__main__.ObjectCreator object at 0x8974f2c>
```

但是在Python中类也是对象.

是的,对象.

每当你用到关键字 `class` ,Python就会执行它并且建立一个对象.例如:

```
>>> class ObjectCreator(object):
...       pass
...
```

上面代码在内存里创建了名叫"ObjectCreator"的对象.

这个对象(类)有生成对象(实例)的能力,这就是为什么叫做类.

它是个对象,所以:

- 你可以把它赋值给一个变量
- 你可以赋值它
- 你可以给它添加属性
- 你个以作为函数参数来传递它

e.g.:

```
>>> print(ObjectCreator) # 你可以打印一个类,因为它是一个对象
<class '__main__.ObjectCreator'>
>>> def echo(o):
...       print(o)
...
>>> echo(ObjectCreator) # 你可以把类作为参数传递
<class '__main__.ObjectCreator'>
>>> print(hasattr(ObjectCreator, 'new_attribute'))
False
>>> ObjectCreator.new_attribute = 'foo' # 可以给一个类添加属性
>>> print(hasattr(ObjectCreator, 'new_attribute'))
True
>>> print(ObjectCreator.new_attribute)
foo
>>> ObjectCreatorMirror = ObjectCreator # 可以把类赋值给一个变量
>>> print(ObjectCreatorMirror.new_attribute)
foo
>>> print(ObjectCreatorMirror())
<__main__.ObjectCreator object at 0x8997b4c>
```

动态创建类

因为类也是对象，你可以在运行时动态的创建它们，就像其他任何对象一样。

首先，你可以在函数中创建类，使用class关键字即可:

```
>>> def choose_class(name):
...     if name == 'foo':
...         class Foo(object):
...             pass
...         return Foo # 返回一个类不是一个实例
...     else:
...         class Bar(object):
...             pass
...         return Bar
...
>>> MyClass = choose_class('foo')
>>> print(MyClass) # 返回一个类不是一个实例
<class '__main__.Foo'>
>>> print(MyClass()) # 你可以在类里创建一个对象
<__main__.Foo object at 0x89c6d4c>
```

但这还不够动态,因为你仍然需要自己编写整个类的代码.

既然类是对象,那么肯定有什么东西来生成它.

当你使用关键字 `objects` ,Python自动的创建对象.像Python中大多数的东西一样,他也给你自己动手的机会.

记得函数 `type` 吗?这个古老好用的函数能让你知道对象的类型是什么:

```
>>> print(type(1))
<type 'int'>
>>> print(type("1"))
<type 'str'>
>>> print(type(ObjectCreator))
<type 'type'>
>>> print(type(ObjectCreator()))
<class '__main__.ObjectCreator'>
```

这里, `type` 有一种完全不同的能力,它也能动态的创建类. `type` 可以接受一个类的描述作为参数,然后返回一个类.

(我知道，根据传入参数的不同，同一个函数拥有两种完全不同的用法是一件很傻的事情，但这在Python中是为了保持向后兼容性)

```
```python
type(类名,
     父类名的元组 (针对继承情况,可以为空),
     包含属性的字典(名称和值))
```

e.g.:

```
>>> class MyShinyClass(object):
...       pass
```

可以手动创建:

```
>>> MyShinyClass = type('MyShinyClass', (), {}) # 返回类对象
>>> print(MyShinyClass)
<class '__main__.MyShinyClass'>
>>> print(MyShinyClass()) # 创建一个类的实例
<__main__.MyShinyClass object at 0x8997cec>
```

你会发现我们使用“MyShinyClass”作为类名，并且也可以把它当做一个变量来作为类的引用。类和变量是不同的，这里没有任何理由把事情弄的复杂。

```
```python
>>> class Foo(object):
...       bar = True
```

可以写成:

```
>>> Foo = type('Foo', (), {'bar':True})
```

然后我们可以像用正常类来用它:

```
>>> print(Foo)
<class '__main__.Foo'>
>>> print(Foo.bar)
True
>>> f = Foo()
>>> print(f)
<__main__.Foo object at 0x8a9b84c>
>>> print(f.bar)
True
```

当然,你也可以继承它:

```
>>>   class FooChild(Foo):
...         pass
```

这样:

```
>>> FooChild = type('FooChild', (Foo,), {})
>>> print(FooChild)
<class '__main__.FooChild'>
>>> print(FooChild.bar) # bar从Foo继承
True
```

要是在类中添加方法,你要做的就是把函数名写入字典就可以了,不懂可以看下面:

```
>>> def echo_bar(self):
...       print(self.bar)
...
>>> FooChild = type('FooChild', (Foo,), {'echo_bar': echo_bar})
>>> hasattr(Foo, 'echo_bar')
False
>>> hasattr(FooChild, 'echo_bar')
True
>>> my_foo = FooChild()
>>> my_foo.echo_bar()
True
```

你可以看到, 在Python中, 类也是对象, 你可以动态的创建类。这就是当你使用关键字class时Python在幕后做的事情, 而这就是通过元类来实现的。

什么是元类(终于到正题了)

元类就是创建类的东西.

你是为了创建对象才定义类的,对吧?

但是我们已经知道了Python的类是对象.

这里,元类创建类.它们是类的类,你可以把它们想象成这样:

```
MyClass = MetaClass()
MyObject = MyClass()
```

你已经看到了 `type` 可以让你像这样做：

```
MyClass = type('MyClass', (), {})
```

这是因为 `type` 就是一个元类. `type` 是Python中创建所有类的元类.

现在你可能纳闷为啥子 `type` 用小写而不写成 `Type` ?

我想是因为要跟 `str` 保持一致, `str` 创建字符串对象, `int` 创建整数对象. `type` 正好创建类对象.

你可以通过检查 `__class__` 属性来看到这一点.

Python中所有的东西都是对象.包括整数,字符串,函数还有类.所有这些都是对象.所有这些也都是从类中创建的:

```
>>> age = 35
>>> age.__class__
<type 'int'>
>>> name = 'bob'
>>> name.__class__
<type 'str'>
>>> def foo(): pass
>>> foo.__class__
<type 'function'>
>>> class Bar(object): pass
>>> b = Bar()
>>> b.__class__
<class '__main__.Bar'>
```

那么, `__class__` 的 `__class__` 属性是什么?

```
>>> age.__class__.__class__
<type 'type'>
>>> name.__class__.__class__
<type 'type'>
>>> foo.__class__.__class__
<type 'type'>
>>> b.__class__.__class__
<type 'type'>
```

所以,元类就是创建类对象的东西.

如果你愿意你也可以把它叫做'类工厂'. `type` 是Python的内建元类,当然,你也可以创建你自己的元类.

`__metaclass__` 属性

当你创建一个函数的时候,你可以添加 `__metaclass__` 属性:

```
class Foo(object):
  __metaclass__ = something...
  [...]
```

如果你这么做了，Python就会用元类来创建类Foo.

小心点，这里面有些技巧.

你首先写下 class Foo(object)，但是类对象 Foo 还没有在内存中创建.

Python将会在类定义中寻找 __metaclass__ .如果找打了就用它来创建类对象 Foo .如果没找到,就会默认用 type 创建类.

把下面这段话反复读几次。

当你写如下代码时 :

```
class Foo(Bar):
  pass
```

Python将会这样运行:

在 Foo 中有没有 ___metaclass__ 属性?

如果有,Python会在内存中通过 __metaclass__ 创建一个名字为 Foo 的类对象(我说的是类对象,跟紧我的思路).

如果Python没有找到 __metaclass__ ，它会继续在Bar（父类）中寻找 __metaclass__属性，并尝试做和前面同样的操作.

如果Python在任何父类中都找不到 __metaclass__ ，它就会在模块层次中去寻找 __metaclass__ ，并尝试做同样的操作。

如果还是找不到 __metaclass__ ,Python就会用内置的 type 来创建这个类对象。

现在的问题就是，你可以在 __metaclass__ 中放置些什么代码呢？

答案就是：可以创建一个类的东西。

那么什么可以用来创建一个类呢？type，或者任何使用到 type 或者子类化 type 的东东都可以。

自定义元类

元类的主要目的就是为了当创建类时能够自动地改变类.

通常，你会为API做这样的事情，你希望可以创建符合当前上下文的类.

假想一个很傻的例子，你决定在你的模块里所有的类的属性都应该是大写形式。有好几种方法可以办到，但其中一种就是通过在模块级别设定 __metaclass__ .

采用这种方法，这个模块中的所有类都会通过这个元类来创建，我们只需要告诉元类把所有的属性都改成大写形式就万事大吉了。

幸运的是，`__metaclass__`实际上可以被任意调用，它并不需要是一个正式的类（我知道，某些名字里带有'class'的东西并不需要是一个class，画画图理解下，这很有帮助）。

所以，我们这里就先以一个简单的函数作为例子开始。

```
# 元类会自动将你通常传给'type'的参数作为自己的参数传入
def upper_attr(future_class_name, future_class_parents, future_class_attr):
  """
    返回一个将属性列表变为大写字母的类对象
  """

  # 选取所有不以'__'开头的属性,并把它们编程大写
  uppercase_attr = {}
  for name, val in future_class_attr.items():
      if not name.startswith('__'):
          uppercase_attr[name.upper()] = val
      else:
          uppercase_attr[name] = val

  # 用'type'创建类
  return type(future_class_name, future_class_parents, uppercase_attr)

__metaclass__ = upper_attr # 将会影响整个模块

class Foo(): # global __metaclass__ won't work with "object" though
  # 我们也可以只在这里定义__metaclass__，这样就只会作用于这个类中
  bar = 'bip'

print(hasattr(Foo, 'bar'))
# 输出: False
print(hasattr(Foo, 'BAR'))
# 输出: True

f = Foo()
print(f.BAR)
# 输出: 'bip'
```

现在让我们再做一次，这一次用一个真正的class来当做元类。

```
# 请记住，'type'实际上是一个类，就像'str'和'int'一样
# 所以，你可以从type继承
class UpperAttrMetaclass(type):
    # __new__ 是在__init__之前被调用的特殊方法
    # __new__是用来创建对象并返回它的方法
    # 而__init__只是用来将传入的参数初始化给对象
    # 你很少用到__new__，除非你希望能够控制对象的创建
    # 这里，创建的对象是类，我们希望能够自定义它，所以我们这里改写__new__
    # 如果你希望的话，你也可以在__init__中做些事情
    # 还有一些高级的用法会涉及到改写__call__特殊方法，但是我们这里不用
    def __new__(upperattr_metaclass, future_class_name,
                future_class_parents, future_class_attr):

        uppercase_attr = {}
        for name, val in future_class_attr.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val

        return type(future_class_name, future_class_parents, uppercase_attr)
```

但是这不是真正的面向对象(OOP).我们直接调用了type，而且我们没有改写父类的**new**方法。现在让我们这样去处理:

```
class UpperAttrMetaclass(type):

    def __new__(upperattr_metaclass, future_class_name,
                future_class_parents, future_class_attr):

        uppercase_attr = {}
        for name, val in future_class_attr.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val

        # 重用 type.__new__ 方法
        # 这就是基本的OOP编程，没什么魔法
        return type.__new__(upperattr_metaclass, future_class_name,
                            future_class_parents, uppercase_attr)
```

你可能已经注意到了有个额外的参数 `upperattr_metaclass`，这并没有什么特别的。类方法的第一个参数总是表示当前的实例，就像在普通的类方法中的 `self` 参数一样。

当然了，为了清晰起见，这里的名字我起的比较长。但是就像 `self` 一样，所有的参数都有它们的传统名称。因此，在真实的产品代码中一个元类应该是像这样的：

```
class UpperAttrMetaclass(type):

    def __new__(cls, clsname, bases, dct):

        uppercase_attr = {}
        for name, val in dct.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val

        return type.__new__(cls, clsname, bases, uppercase_attr)
```

如果使用super方法的话，我们还可以使它变得更清晰一些，这会缓解继承（是的，你可以拥有元类，从元类继承，从type继承）

```
class UpperAttrMetaclass(type):

    def __new__(cls, clsname, bases, dct):

        uppercase_attr = {}
        for name, val in dct.items():
            if not name.startswith('__'):
                uppercase_attr[name.upper()] = val
            else:
                uppercase_attr[name] = val

        return super(UpperAttrMetaclass, cls).__new__(cls, clsname, bases, uppercase_attr)
```

就是这样，除此之外，关于元类真的没有别的可说的了。

使用到元类的代码比较复杂，这背后的原因倒并不是因为元类本身，而是因为你通常会使用元类去做一些晦涩的事情，依赖于自省，控制继承等等。

确实，用元类来搞些“黑暗魔法”是特别有用的，因而会搞出些复杂的东西来。但就元类本身而言，它们其实是很简单的：

- 拦截类的创建
- 修改一个类
- 返回修改之后的类

为什么要用**metaclass**类而不是函数?

由于`__metaclass__`可以接受任何可调用的对象，那为何还要使用类呢，因为很显然使用类会更加复杂啊？

这里有好几个原因：

- 意图会更加清晰。当你读到`UpperAttrMetaclass(type)`时，你知道接下来要发生什么。
- 你可以使用OOP编程。元类可以从元类中继承而来，改写父类的方法。元类甚至还可以使用元类。
- 你可以把代码组织的更好。当你使用元类的时候肯定不会是像我上面举的这种简单场景，通常都是针对比较复杂的问题。将多个方法归总到一个类中会很有帮助，也会使得代码更容易阅读。
- 你可以使用`__new__`, `__init__`以及`__call__`这样的特殊方法。它们能帮你处理不同的任务。就算通常你可以把所有的东西都在`__new__`里处理掉，有些人还是觉得用`__init__`更舒服些。
- 哇哦，这东西的名字是metaclass，肯定非善类，我要小心！

说了这么多TMD究竟为什么要使用元类？

现在回到我们的大主题上来，究竟是为什么你会去使用这样一种容易出错且晦涩的特性？

好吧，一般来说，你根本就用不上它：

> “元类就是深度的魔法，99%的用户应该根本不必为此操心。如果你想搞清楚究竟是否需要用到元类，那么你就不需要它。那些实际用到元类的人都非常清楚地知道他们需要做什么，而且根本不需要解释为什么要用元类。” —— Python界的领袖 Tim Peters

元类的主要用途是创建API。一个典型的例子是Django ORM。

它允许你像这样定义：

```
class Person(models.Model):
  name = models.CharField(max_length=30)
  age = models.IntegerField()
```

但是如果你像这样做的话：

```
guy = Person(name='bob', age='35')
print(guy.age)
```

这并不会返回一个`IntegerField`对象，而是会返回一个int，甚至可以直接从数据库中取出数据。

这是有可能的，因为`models.Model`定义了`__metaclass__`，并且使用了一些魔法能够将你刚刚定义的简单的Person类转变成对数据库的一个复杂hook。

Django框架将这些看起来很复杂的东西通过暴露出一个简单的使用元类的API将其化简，通过这个API重新创建代码，在背后完成真正的工作。

结语

首先，你知道了类其实是能够创建出类实例的对象。

好吧，事实上，类本身也是实例，当然，它们是元类的实例。

```
>>> class Foo(object): pass
>>> id(Foo)
142630324
```

Python中的一切都是对象，它们要么是类的实例，要么是元类的实例.

除了`type`.`type`实际上是它自己的元类，在纯Python环境中这可不是你能够做到的，这是通过在实现层面耍一些小手段做到的。

其次，元类是很复杂的。对于非常简单的类，你可能不希望通过使用元类来对类做修改。你可以通过其他两种技术来修改类：

- monkey patching

- 装饰器

当你需要动态修改类时，99%的时间里你最好使用上面这两种技术。当然了，其实在99%的时间里你根本就不需要动态修改类 :D

rank	▲	✰	vote	url
3	1346	1485	2648	url

Python中如何在一个函数中加入多个装饰器?

怎么做才能让一个函数同时用两个装饰器,像下面这样:

```
@makebold
@makeitalic
def say():
   return "Hello"
```

我希望得到

```
<b><i>Hello</i></b>
```

我只是想知道装饰器怎么工作的!

去看看文档,答在下面:

```
def makebold(fn):
    def wrapped():
        return "<b>" + fn() + "</b>"
    return wrapped

def makeitalic(fn):
    def wrapped():
        return "<i>" + fn() + "</i>"
    return wrapped

@makebold
@makeitalic
def hello():
    return "hello world"

print hello() ## returns <b><i>hello world</i></b>
```

Answer:2

如果你不想看详细的解释的话请看上面那个答案.

装饰器基础

Python的函数都是对象

要了解装饰器,你必须了解Python中的函数都是对象.这个意义非常重要.让我们看看一个简单例子:

```
def shout(word="yes"):
    return word.capitalize()+"!"

print shout()
# 输出 : 'Yes!'

# 作为一个对象,你可以把它赋值给任何变量

scream = shout

# 注意啦我们没有加括号,我们并不是调用这个函数,我们只是把函数"shout"放在了变量"scream"里.
# 也就是说我们可以通过"scream"调用"shout":

print scream()
# 输出 : 'Yes!'

# 你可以删除旧名"shout",而且"scream"依然指向函数

del shout
try:
    print shout()
except NameError, e:
    print e
    #输出: "name 'shout' is not defined"

print scream()
# 输出: 'Yes!'
```

好了,先记住上面的,一会还会用到.

Python函数另一个有趣的特性就是你可以在一个函数里定义另一个函数!

```
def talk():

    # 你可以在"talk"里定义另一个函数 ...
    def whisper(word="yes"):
        return word.lower()+"..."

    # 让我们用用它!

    print whisper()

# 每次调用"talk"时都会定义一次"whisper",然后"talk"会调用"whisper"
talk()
# 输出:
# "yes..."

# 但是在"talk"意外"whisper"是不存在的:

try:
    print whisper()
except NameError, e:
    print e
    #输出 : "name 'whisper' is not defined"*
```

函数引用

好,终于到了有趣的地方了...

已经知道函数就是对象.因此,对象:

- 可以赋值给一个变量
- 可以在其他函数里定义

这就意味着函数可以返回另一个函数.来看看!☺

```
def getTalk(kind="shout"):

    # 在函数里定义一个函数
    def shout(word="yes"):
        return word.capitalize()+"!"

    def whisper(word="yes") :
        return word.lower()+"...";

    # 返回一个函数
    if kind == "shout":
        # 这里不用"()",我们不是要调用函数
        # 只是返回函数对象
        return shout
    else:
        return whisper

# 怎么用这个特性呢?

# 把函数赋值给变量
talk = getTalk()

# 可以看到"talk"是一个函数对象
print talk
# 输出 : <function shout at 0xb7ea817c>

# 函数返回的是对象:
print talk()
# 输出 : Yes!

# 不嫌麻烦你也可以这么用
print getTalk("whisper")()
# 输出 : yes...
```

既然可以 `return` 一个函数, 你也可以在把函数作为参数传递:

```
def doSomethingBefore(func):
    print "I do something before then I call the function you gave me"
    print func()

doSomethingBefore(scream)
# 输出:
#I do something before then I call the function you gave me
#Yes!
```

学习装饰器的基本知识都在上面了.装饰器就是"wrappers",它可以让你在你装饰函数之前或之后执行程序,而不用修改函数本身.

自己动手实现装饰器

怎么样自己做呢:

```
# 装饰器就是把其他函数作为参数的函数
def my_shiny_new_decorator(a_function_to_decorate):

    # 在函数里面,装饰器在运行中定义函数: 包装.
    # 这个函数将被包装在原始函数的外面,所以可以在原始函数之前和之后执行其他代码..
    def the_wrapper_around_the_original_function():

        # 把要在原始函数被调用前的代码放在这里
        print "Before the function runs"

        # 调用原始函数(用括号)
        a_function_to_decorate()

        # 把要在原始函数调用后的代码放在这里
        print "After the function runs"

    # 在这里"a_function_to_decorate" 函数永远不会被执行
    # 在这里返回刚才包装过的函数
    # 在包装函数里包含要在原始函数前后执行的代码.
    return the_wrapper_around_the_original_function

# 加入你建了个函数,不想修改了
def a_stand_alone_function():
    print "I am a stand alone function, don't you dare modify me"

a_stand_alone_function()
#输出: I am a stand alone function, don't you dare modify me

# 现在,你可以装饰它来增加它的功能
# 把它传递给装饰器,它就会返回一个被包装过的函数.

a_stand_alone_function_decorated = my_shiny_new_decorator(a_stand_alone_function)
a_stand_alone_function_decorated()
#输出s:
#Before the function runs
#I am a stand alone function, don't you dare modify me
#After the function runs
```

现在,你或许每次都想用 `a_stand_alone_function_decorated` 代替 `a_stand_alone_function` ,很简单,只需要用 `my_shiny_new_decorator` 返回的函数重写 `a_stand_alone_function` :

```
a_stand_alone_function = my_shiny_new_decorator(a_stand_alone_function)
a_stand_alone_function()
#输出:
#Before the function runs
#I am a stand alone function, don't you dare modify me
#After the function runs

# 想到了吗,这就是装饰器干的事!
```

让我们看看装饰器的真实面纱

用上一个例子,看看装饰器的语法:

```
@my_shiny_new_decorator
def another_stand_alone_function():
    print "Leave me alone"

another_stand_alone_function()
#输出:
#Before the function runs
#Leave me alone
#After the function runs
```

就这么简单. `@decorator` 就是下面的简写:

```
another_stand_alone_function = my_shiny_new_decorator(another_stand_alone_function)
```

装饰器就是 decorator design pattern的pythonic的变种.在Python中有许多经典的设计模式来满足开发者.

当然,你也可以自己写装饰器:

```
def bread(func):
    def wrapper():
        print "</''''''\>"
        func()
        print "<\______/>"
    return wrapper

def ingredients(func):
    def wrapper():
        print "#tomatoes#"
        func()
        print "~salad~"
    return wrapper

def sandwich(food="--ham--"):
    print food

sandwich()
#outputs: --ham--
sandwich = bread(ingredients(sandwich))
sandwich()
#outputs:
#</''''''\>
# #tomatoes#
# --ham--
# ~salad~
#<\______/>
```

用Python装饰器语法糖:

```
@bread
@ingredients
def sandwich(food="--ham--"):
    print food

sandwich()
#outputs:
#</''''''\>
# #tomatoes#
# --ham--
# ~salad~
#<\______/>
```

改变一下顺序:

```
@ingredients
@bread
def strange_sandwich(food="--ham--"):
    print food

strange_sandwich()
#outputs:
##tomatoes#
#</''''''\>
# --ham--
#<\______/>
# ~salad~
```

现在:回答你的问题...

作为结论,相信你现在已经知道答案了:

```
# 字体变粗装饰器
def makebold(fn):
    # 装饰器将返回新的函数
    def wrapper():
        # 在之前或者之后插入新的代码
        return "<b>" + fn() + "</b>"
    return wrapper

# 斜体装饰器
def makeitalic(fn):
    # 装饰器将返回新的函数
    def wrapper():
        # 在之前或者之后插入新的代码
        return "<i>" + fn() + "</i>"
    return wrapper

@makebold
@makeitalic
def say():
    return "hello"

print say()
#输出: <b><i>hello</i></b>

# 这相当于
def say():
    return "hello"
say = makebold(makeitalic(say))

print say()
#输出: <b><i>hello</i></b>
```

别轻松太早,看看下面的高级用法

装饰器高级用法

在装饰器函数里传入参数

```
# 这不是什么黑魔法,你只需要让包装器传递参数:

def a_decorator_passing_arguments(function_to_decorate):
    def a_wrapper_accepting_arguments(arg1, arg2):
        print "I got args! Look:", arg1, arg2
        function_to_decorate(arg1, arg2)
    return a_wrapper_accepting_arguments

# 当你调用装饰器返回的函数时,也就调用了包装器,把参数传入包装器里,
# 它将把参数传递给被装饰的函数里.

@a_decorator_passing_arguments
def print_full_name(first_name, last_name):
    print "My name is", first_name, last_name

print_full_name("Peter", "Venkman")
# 输出:
#I got args! Look: Peter Venkman
#My name is Peter Venkman
```

装饰方法

在Python里方法和函数几乎一样.唯一的区别就是方法的第一个参数是一个当前对象的(self)

也就是说你可以用同样的方式来装饰方法!只要记得把 self 加进去:

```
def method_friendly_decorator(method_to_decorate):
    def wrapper(self, lie):
        lie = lie - 3 # 女性福音 :-)
        return method_to_decorate(self, lie)
    return wrapper


class Lucy(object):

    def __init__(self):
        self.age = 32

    @method_friendly_decorator
    def sayYourAge(self, lie):
        print "I am %s, what did you think?" % (self.age + lie)

l = Lucy()
l.sayYourAge(-3)
#输出: I am 26, what did you think?
```

如果你想造一个更通用的可以同时满足方法和函数的装饰器,用 *args,**kwargs 就可以了

```
def a_decorator_passing_arbitrary_arguments(function_to_decorate):
    # 包装器接受所有参数
    def a_wrapper_accepting_arbitrary_arguments(*args, **kwargs):
        print "Do I have args?:"
        print args
        print kwargs
        # 现在把*args,**kwargs解包
        # 如果你不明白什么是解包的话,请查阅:
        # http://www.saltycrane.com/blog/2008/01/how-to-use-args-and-kwargs-in-python/
        function_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_arguments

@a_decorator_passing_arbitrary_arguments
def function_with_no_argument():
    print "Python is cool, no argument here."

function_with_no_argument()
#输出
#Do I have args?:
#()
#{}
#Python is cool, no argument here.

@a_decorator_passing_arbitrary_arguments
def function_with_arguments(a, b, c):
    print a, b, c

function_with_arguments(1,2,3)
#输出
#Do I have args?:
#(1, 2, 3)
```

```
#{}
#1 2 3

@a_decorator_passing_arbitrary_arguments
def function_with_named_arguments(a, b, c, platypus="Why not ?"):
    print "Do %s, %s and %s like platypus? %s" %\
    (a, b, c, platypus)

function_with_named_arguments("Bill", "Linus", "Steve", platypus="Indeed!")
#输出
#Do I have args ? :
#('Bill', 'Linus', 'Steve')
#{'platypus': 'Indeed!'}
#Do Bill, Linus and Steve like platypus? Indeed!

class Mary(object):

    def __init__(self):
        self.age = 31

    @a_decorator_passing_arbitrary_arguments
    def sayYourAge(self, lie=-3): # 可以加入一个默认值
        print "I am %s, what did you think ?" % (self.age + lie)

m = Mary()
m.sayYourAge()
#输出
# Do I have args?:
#(<__main__.Mary object at 0xb7d303ac>,)
#{}
#I am 28, what did you think?
```

把参数传递给装饰器

好了,如何把参数传递给装饰器自己?

因为装饰器必须接收一个函数当做参数,所以有点麻烦.好吧,你不可以直接把被装饰函数的参数传递给装饰器.

在我们考虑这个问题时,让我们重新回顾下:

```python
# 装饰器就是一个'平常不过'的函数
def my_decorator(func):
    print "I am an ordinary function"
    def wrapper():
        print "I am function returned by the decorator"
        func()
    return wrapper

# 因此你可以不用"@"也可以调用他

def lazy_function():
    print "zzzzzzzz"

decorated_function = my_decorator(lazy_function)
#输出: I am an ordinary function

# 之所以输出 "I am an ordinary function"是因为你调用了函数,
# 并非什么魔法.

@my_decorator
def lazy_function():
    print "zzzzzzzz"

#输出: I am an ordinary function
```

看见了吗,和" `my_decorator` "一样只是被调用.所以当你用 `@my_decorator` 你只是告诉Python去掉用被变量 `my_decorator` 标记的函数.

这非常重要!你的标记能直接指向装饰器.

让我们做点邪恶的事.☺

```
def decorator_maker():

    print "I make decorators! I am executed only once: "+\
          "when you make me create a decorator."

    def my_decorator(func):

        print "I am a decorator! I am executed only when you decorate a function."

        def wrapped():
            print ("I am the wrapper around the decorated function. "
                  "I am called when you call the decorated function. "
                  "As the wrapper, I return the RESULT of the decorated function.")
            return func()

        print "As the decorator, I return the wrapped function."

        return wrapped

    print "As a decorator maker, I return a decorator"
    return my_decorator

# 让我们建一个装饰器.它只是一个新函数.
new_decorator = decorator_maker()
#输出:
#I make decorators! I am executed only once: when you make me create a decorator.
#As a decorator maker, I return a decorator

# 下面来装饰一个函数

def decorated_function():
    print "I am the decorated function."

decorated_function = new_decorator(decorated_function)
#输出:
#I am a decorator! I am executed only when you decorate a function.
#As the decorator, I return the wrapped function

# Let’s call the function:
decorated_function()
#输出:
#I am the wrapper around the decorated function. I am called when you call the decorated function.
#As the wrapper, I return the RESULT of the decorated function.
#I am the decorated function.
```

一点都不难把.

下面让我们去掉所有可恶的中间变量:

```
def decorated_function():
    print "I am the decorated function."
decorated_function = decorator_maker()(decorated_function)
#输出:
#I make decorators! I am executed only once: when you make me create a decorator.
#As a decorator maker, I return a decorator
#I am a decorator! I am executed only when you decorate a function.
#As the decorator, I return the wrapped function.

# 最后:
decorated_function()
#输出:
#I am the wrapper around the decorated function. I am called when you call the decorated function.
#As the wrapper, I return the RESULT of the decorated function.
#I am the decorated function.
```

让我们简化一下:

```
@decorator_maker()
def decorated_function():
    print "I am the decorated function."
#输出:
#I make decorators! I am executed only once: when you make me create a decorator.
#As a decorator maker, I return a decorator
#I am a decorator! I am executed only when you decorate a function.
#As the decorator, I return the wrapped function.

#最终:
decorated_function()
#输出:
#I am the wrapper around the decorated function. I am called when you call the decorated function.
#As the wrapper, I return the RESULT of the decorated function.
#I am the decorated function.
```

看到了吗?我们用一个函数调用"`@`"语法!:-)

所以让我们回到装饰器的.如果我们在函数运行过程中动态生成装饰器,我们是不是可以把参数传递给函数?

```
def decorator_maker_with_arguments(decorator_arg1, decorator_arg2):

    print "I make decorators! And I accept arguments:", decorator_arg1, decorator_arg2

    def my_decorator(func):
        # 这里传递参数的能力是借鉴了 closures.
        # 如果对closures感到困惑可以看看下面这个:
        # http://stackoverflow.com/questions/13857/can-you-explain-closures-as-they-relate-to-python
        print "I am the decorator. Somehow you passed me arguments:", decorator_arg1, decorator_arg2

        # 不要忘了装饰器参数和函数参数!
        def wrapped(function_arg1, function_arg2) :
            print ("I am the wrapper around the decorated function.\n"
                  "I can access all the variables\n"
                  "\t- from the decorator: {0} {1}\n"
                  "\t- from the function call: {2} {3}\n"
                  "Then I can pass them to the decorated function"
                  .format(decorator_arg1, decorator_arg2,
                          function_arg1, function_arg2))
            return func(function_arg1, function_arg2)

        return wrapped

    return my_decorator

@decorator_maker_with_arguments("Leonard", "Sheldon")
def decorated_function_with_arguments(function_arg1, function_arg2):
    print ("I am the decorated function and only knows about my arguments: {0}"
           " {1}".format(function_arg1, function_arg2))

decorated_function_with_arguments("Rajesh", "Howard")
#输出:
#I make decorators! And I accept arguments: Leonard Sheldon
#I am the decorator. Somehow you passed me arguments: Leonard Sheldon
#I am the wrapper around the decorated function.
#I can access all the variables
#   - from the decorator: Leonard Sheldon
#   - from the function call: Rajesh Howard
#Then I can pass them to the decorated function
#I am the decorated function and only knows about my arguments: Rajesh Howard
```

好了,上面就是带参数的装饰器.参数可以设置成变量:

```
c1 = "Penny"
c2 = "Leslie"

@decorator_maker_with_arguments("Leonard", c1)
def decorated_function_with_arguments(function_arg1, function_arg2):
    print ("I am the decorated function and only knows about my arguments:"
           " {0} {1}".format(function_arg1, function_arg2))

decorated_function_with_arguments(c2, "Howard")
#输出:
#I make decorators! And I accept arguments: Leonard Penny
#I am the decorator. Somehow you passed me arguments: Leonard Penny
#I am the wrapper around the decorated function.
#I can access all the variables
#   - from the decorator: Leonard Penny
#   - from the function call: Leslie Howard
#Then I can pass them to the decorated function
#I am the decorated function and only knows about my arguments: Leslie Howard
```

你可以用这个小技巧把任何函数的参数传递给装饰器.如果你愿意还可以用 `*args,**kwargs` .但是一定要记住了装饰器只能被调用一次.当Python载入脚本后,你不可以动态的设置参数了.当你运行 `import x` ,函数已经被装饰,所以你什么都不能动了.

来练习一下:装饰装饰器

好吧,作为奖励,我就给你讲讲如何怎么让所有的装饰器接收任何参数.为了接收参数,我们用另外的函数来建我们的装饰器.

我们包装装饰器.

还有什么我们可以看到吗?

对了,装饰器!

让我们来为装饰器一个装饰器:

```python
def decorator_with_args(decorator_to_enhance):
    """
    这个函数将被用来作为装饰器.
    它必须去装饰要成为装饰器的函数.
    休息一下.
    它将允许所有的装饰器可以接收任意数量的参数,所以以后你不必为每次都要做这个头疼了.
    saving you the headache to remember how to do that every time.
    """

    # 我们用传递参数的同样技巧.
    def decorator_maker(*args, **kwargs):

        # 我们动态的建立一个只接收一个函数的装饰器,
        # 但是他能接收来自maker的参数
        def decorator_wrapper(func):

            # 最后我们返回原始的装饰器,毕竟它只是'平常'的函数
            # 唯一的陷阱:装饰器必须有这个特殊的,否则将不会奏效.
            return decorator_to_enhance(func, *args, **kwargs)

        return decorator_wrapper
```

下面是如何用它们:

```python
# 下面的函数是你建来当装饰器用的,然后把装饰器加到上面:-)
# 不要忘了这个 "decorator(func, *args, **kwargs)"
@decorator_with_args
def decorated_decorator(func, *args, **kwargs):
    def wrapper(function_arg1, function_arg2):
        print "Decorated with", args, kwargs
        return func(function_arg1, function_arg2)
    return wrapper

# 现在你用你自己的装饰装饰器来装饰你的函数(汗~~~)

@decorated_decorator(42, 404, 1024)
def decorated_function(function_arg1, function_arg2):
    print "Hello", function_arg1, function_arg2

decorated_function("Universe and", "everything")
#输出:
#Decorated with (42, 404, 1024) {}
#Hello Universe and everything

# Whoooot!
```

估计你看到这和你刚看完爱因斯坦相对论差不多,但是现在如果明白怎么用就好多了吧.

最好的练习:装饰器

- 装饰器是Python2.4里引进的,所以确保你的Python解析器的版本>=2.4
- 装饰器使函数调用变慢了.一定要记住.
- 装饰器不能被取消(有些人把装饰器做成可以移除的但是没有人会用)所以一旦一个函数被装饰了.所有的代码都会被装饰.
- 用装饰器装饰函数将会很难debug(在>=2.5版本将会有所改善;看下面)

functools 模块在2.5被引进.它包含了一个 functools.wraps() 函数,可以复制装饰器函数的名字,模块和文档给它的包装器.

(事实上: functools.wraps() 是一个装饰器! ☺)

```
#为了debug,堆栈跟踪将会返回函数的 __name__
def foo():
    print "foo"

print foo.__name__
#输出: foo

# 如果加上装饰器,将变得有点复杂
def bar(func):
    def wrapper():
        print "bar"
        return func()
    return wrapper

@bar
def foo():
    print "foo"

print foo.__name__
#输出: wrapper

# "functools" 将有所帮助

import functools

def bar(func):
    # 我们所说的"wrapper",正在包装 "func",
    # 好戏开始了
    @functools.wraps(func)
    def wrapper():
        print "bar"
        return func()
    return wrapper

@bar
def foo():
    print "foo"

print foo.__name__
#输出: foo
```

怎么使用装饰器?

现在遇到了大问题:我们用装饰器干什么?

看起来很黄很暴力,但是如果有实际用途就更好了.好了这里有1000个用途.传统的用法就是用它来为外部的库的函数(你不能修改的)做扩展,或者debug(你不想修改它,因为它是暂时的).

你也可以用DRY的方法去扩展一些函数,像:

```
def benchmark(func):
    """
    A decorator that prints the time a function takes
    to execute.
    """
    import time
    def wrapper(*args, **kwargs):
        t = time.clock()
        res = func(*args, **kwargs)
        print func.__name__, time.clock()-t
        return res
    return wrapper


def logging(func):
    """
    A decorator that logs the activity of the script.
    (it actually just prints it, but it could be logging!)
    """
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs)
        print func.__name__, args, kwargs
        return res
    return wrapper


def counter(func):
    """
    A decorator that counts and prints the number of times a function has been executed
    """
    def wrapper(*args, **kwargs):
        wrapper.count = wrapper.count + 1
        res = func(*args, **kwargs)
        print "{0} has been used: {1}x".format(func.__name__, wrapper.count)
        return res
    wrapper.count = 0
    return wrapper

@counter
@benchmark
@logging
def reverse_string(string):
    return str(reversed(string))

print reverse_string("Able was I ere I saw Elba")
print reverse_string("A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps, snipe, percale, mac

#输出:
#reverse_string ('Able was I ere I saw Elba',) {}
#wrapper 0.0
#wrapper has been used: 1x
#ablE was I ere I saw elbA
#reverse_string ('A man, a plan, a canoe, pasta, heros, rajahs, a coloratura, maps, snipe, percale, macaro
#wrapper 0.0
#wrapper has been used: 2x
#!amanaP :lanac a ,noep a ,stah eros ,raj a ,hsac ,oloR a ,tur a ,mapS ,snip ,eperc a ,)lemac a ro( niaga ga
```

当然,装饰器的好处就是你可以用它们来做任何事而不用重写,DRY:

```
@counter
@benchmark
@logging
def get_random_futurama_quote():
    from urllib import urlopen
    result = urlopen("http://subfusion.net/cgi-bin/quote.pl?quote=futurama").read()
    try:
        value = result.split("<br><b><hr><br>")[1].split("<br><br><hr>")[0]
        return value.strip()
    except:
        return "No, I'm ... doesn't!"


print get_random_futurama_quote()
print get_random_futurama_quote()

#输出:
#get_random_futurama_quote () {}
#wrapper 0.02
#wrapper has been used: 1x
#The laws of science be a harsh mistress.
#get_random_futurama_quote () {}
#wrapper 0.01
#wrapper has been used: 2x
#Curse you, merciful Poseidon!
```

Python自身提供了几个装饰器,像`property`, `staticmethod`.

- Django用装饰器管理缓存和试图的权限.
- Twisted用来修改异步函数的调用.

好大的坑!

rank	▲	☆	vote	url
4	1266	285	929	url

用**Python**如何一个文件是否存在**?**

不用`try:`语句可以一个文件存在

如果不确定文件存不存在,可以这样做:

```
import os.path
os.path.isfile(fname)
```

rank	▲	☆	vote	url
5	1187	203	1370	url

在Python中有三元运算符吗?

如果没有,可以像其他语言用的简单方法来实现吗?

有,在2.5版本中加入.对于python初学者可能有点难以理解,所以要记住了.

语法如下:

```
a if test else b
```

根据`test`的布尔值来判断返回的是`a`还是`b`;如果`test`为真则返回`a`,反之则返回`b`.

来个大栗子:

```
>>> 'true' if True else 'false'
'true'
>>> 'true' if False else 'false'
'false'
```

官方文档:

- Conditional expressions
- Is there an equivalent of C’s ”?:” ternary operator?

rank	▲	☆	vote	url
6	1161	520	1167	url

在Python中调用外部命令?

怎么在Python脚本里调用外部命令?(就好像直接输入在Unix shell中或者windows的命令行里)

来来来,我给你叨咕叨咕各种方法和各自优缺点:

1. `os.system("命令加参数")` 把命令和参数传递给你系统的shell中.用这个命令的好处在于你可以一次运行好多命令还可以设置管道来进行重定向.来个栗子: `os.system("命令 < 出入文件 | 另一个命令 > 输出文件")` 尽管它非常方便,但是你还是不得不手动输入像空格这样的sehll字符.从另一方面讲,对于运行简单的shell命令而不去调用外部程序来说的话还是非常好用的.
2. `stream = os.popen("命令和参数")` 这个命令和 `os.system` 差不多,但是它提供了一个连接标准输入/输出的管道.还有其他3个 `popen` 可以调用.如果你传递一个字符串,你的命令会把它传递给shell,如果你传递的是一个列表,那么就不用担心溢出字符了(escaping characters).
3. `subprocess` 模块的管道 `Popen` .这个 `Popen` 是打算用来替代 `os.popen` 的方法,它有点复杂:

   ```
   print subprocess.Popen("echo Hello World",     shell=True,stdout=PIPE).stdout.read()
   ```

   而用 `os.popen` :

   ```
   print os.popen("echo Hello World").read()
   ```

   它最大的优点就是一个类里代替了原来的4个不同的 `popen`

4. `subprocess` 的 `call` 方法.它的基本用法和上面的 `Popen` 类参数一致,但是它会等待命令结束后才会返回程序.来个大狸子:

   ```
   return_code = subprocess.call("echo Hello World", shell=True)
   ```

5. os模块里也有C语言里 `fork/exec/spawn` 方法,但是我不建议你直接用它们.

`subprocess` 模块可能更适合你.

最后请注意在你传递到shell的命令一定要注意参数的安全性,给你个提示,看下面代码

```
print subprocess.Popen("echo %s " % user_input, stdout=PIPE).stdout.read()
```

想象一下如果哪个SB输入 `my mama didnt love me && rm -rf /`

rank	▲	☆	vote	url
7	1112	431	1201	url

在Python里如何用枚举类型?

我是一个C#开发者,但是我现在做的工作是关于Python的.

怎么在Python里代替枚举类型呢?

PEP435标准里已经把枚举添加到Python3.4版本,在Pypi中也可以向后支持3.3, 3.2, 3.1, 2.7, 2.6, 2.5, 和2.4版本.

如果想向后兼容 `$ pip install enum34` ,如果下载 `enum` (没有数字)将会是另一个版本.

```
from enum import Enum
Animal = Enum('Animal', 'ant bee cat dog')
```

或者等价于:

```
class Animals(Enum):
    ant = 1
    bee = 2
    cat = 3
    dog = 4
```

在更早的版本,下面这种方法来完成枚举:

```
def enum(**enums):
    return type('Enum', (), enums)
```

像这样来用:

```
>>> Numbers = enum(ONE=1, TWO=2, THREE='three')
>>> Numbers.ONE
1
>>> Numbers.TWO
2
>>> Numbers.THREE
'three'
```

也很容易支持自动计数,像下面这样:

```
def enum(*sequential, **named):
    enums = dict(zip(sequential, range(len(sequential))), **named)
    return type('Enum', (), enums)
```

这样用:

```
>>> Numbers = enum('ZERO', 'ONE', 'TWO')
>>> Numbers.ZERO
0
>>> Numbers.ONE
1
```

如果要把值转换为名字可以加入下面的方法:

```
def enum(*sequential, **named):
    enums = dict(zip(sequential, range(len(sequential))), **named)
    reverse = dict((value, key) for key, value in enums.iteritems())
    enums['reverse_mapping'] = reverse
    return type('Enum', (), enums)
```

这样会覆盖名字下的所有东西,但是对于枚举的输出很有用.如果转换的值不存在就会抛出`KeyError`异常.用前面的例子:

```
>>> Numbers.reverse_mapping['three']
'THREE'
```

rank	▲	☆	vote	url
8	919	442	608	url

怎么在windows下安装pip?

怎么在windows下安装pip?

Python3.4+

好消息,Python3.4已经自带Pip.这是所有Python发行版中最好的特性了.方便了所有人使用公共库.新手再也不用操心安装额外库的繁琐步骤了.它自带的包管理器中加入了 Ruby, Nodejs, Haskell, Perl, Go等其他几乎所有的开源社区流行语言.谢谢Python.

当然,这并不意味着所有的Python包问题已经解决.在一段时间看来仍然不乐观.我在Python有包管理系统吗?也讨论过这个问题.

同样对于Python2.x用户(几乎一般人),还没有计划在Python中自带Pip. 只能自己动手了.

Python 2.x 和 Python ≤ 3.3

尽管Python吹的`简单易用`的哲学,但是Python不提供包管理工具.更糟糕的是,Pip直到现在还是非常的难以安装.

官方指南

在 http://www.pip-installer.org/en/latest/installing.html 下载`get-pip.py`,把它保存下来注意不要把`.py`后缀改成`.txt`. 然后在命令提示符上输入:

```
python get-pip.py
```

你可能还需要管理员权限来执行它.跟着做 http://technet.microsoft.com/en-us/library/cc947813(v=ws.10).aspx

另一种方法

官方文档告诉我们安装Pip和各种依赖的源.对有经验的人来说太麻烦了,对于新手又有点难.

Christoph Gohlke已经为我们做好了下载Python包的安装器(.msi).它可以为的Python版本建立依赖,不管32bit还是64bit.你只需要:

- 安装setuptools http://www.lfd.uci.edu/~gohlke/pythonlibs/#setuptools
- 安装pip http://www.lfd.uci.edu/~gohlke/pythonlibs/#pip

在我这,Pip安装到了`C:\Python27\Scripts\pip.exe`.在你的电脑上找到`pip.exe`,然后把文件夹(eg. `C:\Python27\Scripts`)添加你的路径(编辑你的环境变量).现在你应当可以在命令行中运行pip了.试试安装一个包:

```
pip install httpie
```

终于可以运行了!下面是一些问题的解决办法

代理问题

如果你在办公环境,那么你有可能有一个HTTP代理.把环境变量设置成`http_proxy`和`https_proxy`.大多数应用程序都管用(包括免费软件).语法如下:

```
http://proxy_url:port
http://username:password@proxy_url:port
```

如果你不幸用的是微软的NTLM代理.那就没救了.唯一的办法就是安装一个友好的代理吧.
http://cntlm.sourceforge.net/

找不到**vcvarsall.bat**

Python有的模块是用C或C++写的.Pip尝试从源码编译.如果你没有安装或设置过C/C++编译器,你将会看到下面的错误:

```
Error: Unable to find vcvarsall.bat
```

你可以通过像MinGw或者Visual C++这样的C++编译器来解决此问题.微软实际上已经自带了一个为Python准备的编译器,或者试试 http://aka.ms/vcpython27

可以这个Christoph的网站来查看安装包 http://www.lfd.uci.edu/~gohlke/pythonlibs/

rank	▲	☆	vote	url
9	909	224	1035	url

如何在一个表达式里合并两个字典?

我有两个Python字典,我想写一个表达式来返回两个字典的合并.update()方法返回的是空值而不是返回合并后的对象.

```
>>> x = {'a':1, 'b': 2}
>>> y = {'b':10, 'c': 11}
>>> z = x.update(y)
>>> print z
None
>>> x
{'a': 1, 'b': 10, 'c': 11}
```

怎么样才能最终让值保存在z而不是x?

可以用下面的方法:

```
z = dict(x.items() + y.items())
```

最后就是你想要的最终结果保存在字典z中,而键 `b` 的值会被第二个字典的值覆盖.

```
>>> x = {'a':1, 'b': 2}
>>> y = {'b':10, 'c': 11}
>>> z = dict(x.items() + y.items())
>>> z
{'a': 1, 'c': 11, 'b': 10}
```

如果你用的是Python3的话稍微有点麻烦:

```
>>> z = dict(list(x.items()) + list(y.items()))
>>> z
{'a': 1, 'c': 11, 'b': 10}
```

还可以这样:

```
z = x.copy()
z.update(y)
```

rank	▲	☆	vote	url
10	903	326	325	url

有方法让Python运行在Android上吗?

我喜欢Android平台.试试上我和几个朋友在Spoxt项目里在用ADC(数模转换器?)

但是我一点也不喜欢Java.我们工作在S60版本而且一个不错的Python的API.我知道Android上没有官方的Python版本,但是既然有Jython,有没有什么方法让它们能在Android上工作?

有一种方法,使用Kivy:

> 交互界面快速开发应用的Python开源库,像多点触控app.

> Kivy运行在 Linux, Windows, OS X, Android and iOS.你也可以在所有的平台上运行Python代码

Kivy的应用

rank	▲	☆	vote	url
11	867	379	1107	url

用字典的值对字典进行排序

我有个字典,字典的值来自于数据库:一个是字符串,一个是数字.字符串是唯一的,所以键就是字符串.

我可以用键来排序,但是怎么用值来排序呢?

注:我已经看过另一个问题怎样对列表中的字典的键值对字典进行排序?,或许这种方法可以,但是我确实只需要一个字典,我想看看还有其他更好的方法.

对字典进行排序是不可能的,只有把字典转换成另一种方式才能排序.字典本身是无序的,但是像列表元组等其他类型是有序的.所以你需要用一个元组列表来表示排序的字典.

例子:

```
import operator
x = {1: 2, 3: 4, 4:3, 2:1, 0:0}
sorted_x = sorted(x.items(), key=operator.itemgetter(1))
```

`sorted_x` 是一个元组列表,用每个元组的第二个元素进行排序. `dict(sorted_x) == x` .

如果想要用键来进行排序:

```
import operator
x = {1: 2, 3: 4, 4:3, 2:1, 0:0}
sorted_x = sorted(x.items(), key=operator.itemgetter(0))
```

你也可以:

```
sorted(d.items(), key=lambda x: x[1])
```

rank	▲	☆	vote	url
12	851	228	1287	url

如何在一个函数里用全局变量?

如果我在一个函数里建了一个全局变量,那么我怎么在另一个函数里使用这个全局变量?

我需要把这个全局变量赋值给这个函数的局部变量吗?

如果你要在别的函数里使用全局变量,只要在被调用全局变量函数的里事先用 `global` 声明一下:

```
globvar = 0

def set_globvar_to_one():
    global globvar    # 需要用global修饰一下globvar
    globvar = 1

def print_globvar():
    print globvar     # 如果要读globbar的值的话不需要用global修饰

set_globvar_to_one()
print_globvar()       # 输出 1
```

我猜正是因为全局变量比较危险,所以Python为了确保你真的知道它是全局变量,所以需要加一个 `global` 关键字.

rank	▲	✰	vote	url
13	811	323	639	url

Python中的"小震撼":变化的默认参数

许多使用很长时间Python的人也被下面的问题困扰:

```
def foo(a=[]):
    a.append(5)
    return a
```

Python新手估计可能会想这个函数返回一个只有元素`[5]`的列表.但是结果却出人意料:

```
>>> foo()
[5]
>>> foo()
[5, 5]
>>> foo()
[5, 5, 5]
>>> foo()
[5, 5, 5, 5]
>>> foo()
```

我的一个经理曾经碰到过这个特性并把它叫做语言的"动态设计缺陷".这个现象应当有更深层次的解释,如果你不懂它的内部它确实非常令人困惑.然而我不能回答下面的问题:是什么原因使默认参数在函数的定义时被绑定,而不是在执行时?我怀疑这个特性在现实中有没有实际的用途(就像在C语言中有谁去用静态变量?)

事实上这并不是设计缺陷,也不是什么内部或性能原因.

原因很简单,Python中的函数是最高等级的对象,而不仅仅是一小段代码.

试着这么来理解:一个函数是一个被它自己定义而执行的对象;默认参数是一种**"成员数据",**所以它们的状态和其他对象一样**,**会随着每一次调用而改变**.**

Effbot在它的Python中的默认参数对这种行为的原因解释的非常清楚!我强烈建议你读一读能对函数对象的工作有更深一步的了解.

rank	▲	✰	vote	url
14	805	326	554	url

装饰器@staticmethod和@classmethod有什么区别?

也许一些例子会有帮助:注意 foo, class_foo 和 static_foo 参数的区别:

```
class A(object):
    def foo(self,x):
        print "executing foo(%s,%s)"%(self,x)

    @classmethod
    def class_foo(cls,x):
        print "executing class_foo(%s,%s)"%(cls,x)

    @staticmethod
    def static_foo(x):
        print "executing static_foo(%s)"%x

a=A()
```

下面是一个对象实体调用方法的常用方式.对象实体 a 被隐藏的传递给了第一个参数.

```
a.foo(1)
# executing foo(<__main__.A object at 0xb7dbef0c>,1)
```

用**classmethods**装饰,隐藏的传递给第一个参数的是对象实体的类(class A)而不是 self .

```
a.class_foo(1)
# executing class_foo(<class '__main__.A'>,1)
```

你也可以用类调用 class_foo .实际上,如果你把一些方法定义成 classmethod ,那么实际上你是希望用类来调用这个方法,而不是用这个类的实例来调用这个方法. A.foo(1) 将会返回一个 TypeError 错误, A.class_foo(1) 将会正常运行:

```
A.class_foo(1)
# executing class_foo(<class '__main__.A'>,1)
```

One use people have found for class methods is to create inheritable alternative constructors.

用**staticmethods**来装饰,不管传递给第一个参数的是 self (对象实体)还是 cls (类).它们的表现都一样:

```
a.static_foo(1)
# executing static_foo(1)

A.static_foo('hi')
# executing static_foo(hi)
```

静态方法被用来组织类之间有逻辑关系的函数.

`foo`只是个函数,但是当你调用`a.foo`的时候你得到的不仅仅是一个函数,你得到的是一个第一个参数绑定到`a`的"加强版"函数. `foo`需要两个参数,而`a.foo`仅仅需要一个参数.

`a`绑定了`foo`.下面可以知道什么叫"绑定"了:

```
print(a.foo)
# <bound method A.foo of <__main__.A object at 0xb7d52f0c>>
```

如果使用`a.class_foo`,是`A`绑定到了`class_foo`而不是`a`.

```
print(a.class_foo)
# <bound method type.class_foo of <class '__main__.A'>>
```

最后剩下静态方法,说到底它就是一个方法. `a.static_foo`只是返回一个不带参数绑定的方法. `static_foo`和`a.static_foo`只需要一个参数.

```
print(a.static_foo)
# <function static_foo at 0xb7d479cc>
```

rank	▲	✰	vote	url
15	789	144	1269	url

检查列表是否为空的最好方法

例如,传递下面:

```
a = []
```

我怎么检查`a`是空值?

```
if not a:
  print "List is empty"
```

用隐藏的空列表的布尔值才是最Pythonic的方法.

rank	▲	☆	vote	url
16	789	431	938	url

怎么用引用来改变一个变量?

Python的文档对参数传递的是值还是引用没有明确说明,下面的代码没有改变值 `Original`

```
class PassByReference:
    def __init__(self):
        self.variable = 'Original'
        self.Change(self.variable)
        print self.variable

    def Change(self, var):
        var = 'Changed'
```

有什么方法能让通过引用来改变变量吗?

参数是通过assignment来传递的.原因是双重的:

1. 传递的参数实际上是一个对象的引用(但是这个引用是通过值传递的)
2. 一些数据类型是可变的,但有一些就不是.

所以:

- 如果传递一个可变对象到一个方法,方法就会获得那个对象的引用,而你也可以随心所欲的改变它了.但是你在方法里重新绑定了这个引用,外部是无法得知的,而当函数完成后,外界的引用依然指向原来的对象.
- 如果你传递一个不可变的对象到一个方法,你仍然不能在外边重新绑定引用,你连改变对象都不可以.

为了弄懂,来几个例子.

列表-可变类型

让我们试着修改当做参数传递给方法的列表:

```
def try_to_change_list_contents(the_list):
    print 'got', the_list
    the_list.append('four')
    print 'changed to', the_list

outer_list = ['one', 'two', 'three']

print 'before, outer_list =', outer_list
try_to_change_list_contents(outer_list)
print 'after, outer_list =', outer_list
```

输出:

```
before, outer_list = ['one', 'two', 'three']
got ['one', 'two', 'three']
changed to ['one', 'two', 'three', 'four']
after, outer_list = ['one', 'two', 'three', 'four']
```

因为传递的参数是 `outer_list` 的引用,而不是副本,所以我们可以用可变列表的方法来改变它,而且改变同时反馈到了外部.

现在让我们来看看我们试着改变作为传递参数的引用时到底发生了什么:

```
def try_to_change_list_reference(the_list):
    print 'got', the_list
    the_list = ['and', 'we', 'can', 'not', 'lie']
    print 'set to', the_list

outer_list = ['we', 'like', 'proper', 'English']

print 'before, outer_list =', outer_list
try_to_change_list_reference(outer_list)
print 'after, outer_list =', outer_list
```

输出:

```
before, outer_list = ['we', 'like', 'proper', 'English']
got ['we', 'like', 'proper', 'English']
set to ['and', 'we', 'can', 'not', 'lie']
after, outer_list = ['we', 'like', 'proper', 'English']
```

既然 `the_list` 参数是通过值进行传递的,那么为它赋值将会对方法以外没有影响. `the_list` 是 `outer_list` 引用(注意,名词)的一个拷贝,我们将 `the_list` 指向一个新的列表,但是并没有改变 `outer_list` 的指向.

字符串-不可变类型

它是不可变类型,所以我们不能改变字符串里的内容.

现在,让我们试着改变引用

```
def try_to_change_string_reference(the_string):
    print 'got', the_string
    the_string = 'In a kingdom by the sea'
    print 'set to', the_string

outer_string = 'It was many and many a year ago'

print 'before, outer_string =', outer_string
try_to_change_string_reference(outer_string)
print 'after, outer_string =', outer_string
```

输出:

```
before, outer_string = It was many and many a year ago
got It was many and many a year ago
set to In a kingdom by the sea
after, outer_string = It was many and many a year ago
```

又一次,既然 the_string 参数用值进行传递,对它进行赋值并不能改变方法外的值. the_string 只是 outer_string 引用(名词)的副本,所以我们让 the_string 指向一个新字符串,依然无法改变 outer_string 的指向.

希望你清楚以上那些.

修改:到现在位置还没有回答"有什么方法同过引用传递变量?",让我们往下看.

我们该怎么办?

你可以返回一个新值.这不会改变传过来的值,但是能得到你想要的结果.

```
def return_a_whole_new_string(the_string):
    new_string = something_to_do_with_the_old_string(the_string)
    return new_string

# 你可以像这样调用
my_string = return_a_whole_new_string(my_string)
```

如果你真的不想用一个返回值,你可以建一个存放你的值的类,然后把它传递给函数或者用一个已有的类,像列表:

```
def use_a_wrapper_to_simulate_pass_by_reference(stuff_to_change):
    new_string = something_to_do_with_the_old_string(stuff_to_change[0])
    stuff_to_change[0] = new_string

# 你可以像这样调用
wrapper = [my_string]
use_a_wrapper_to_simulate_pass_by_reference(wrapper)

do_something_with(wrapper[0])
```

虽然看起来有一点笨重,但还是达到你的效果了.

rank	▲	✰	vote	url
17	776	241	907	url

检查一个文件夹是否存在,如果不存在就创建它

有什么好的方法吗?这个是我想的:

```
filename = "/my/directory/filename.txt"
dir = os.path.dirname(filename)

try:
    os.stat(dir)
except:
    os.mkdir(dir)

f = file(filename)
```

我忘记了 `os.path.exists` (多谢张三,李四,王五的提醒).下面是更改的:

```
def ensure_dir(f):
    d = os.path.dirname(f)
    if not os.path.exists(d):
        os.makedirs(d)
```

有什么"打开"的标记可以自动的运行?

我看了俩答案都很好,但是都有一点缺陷,所以给出我的:

先试 `os.path.exists` ,然后通过 `os.makedirs` 来创建.

```
if not os.path.exists(directory):
    os.makedirs(directory)
```

标注一下-如果一个文件在调用 `os.path.exists` 和 `os.makedirs` 之间被创建了,将会出现一个 `OSError` .遗憾的是捕获 `OSError` 异常继续进行并不是万无一失的,它将会忽略像磁盘空间不足,没有足够权限等一些其他造成文件创建失败的因素.

一个做法是捕获 `OSError` 异常并检查返回的错误代码(前提是知道错误代码对应的是什么).然而,还有另一种可能,第二次的 `os.path.exists` .假如恰好在第一次检查的时候创建了文件夹,然后在第二次检查的时候删掉--我们被耍了~~

根据不同的应用,并行操作的危险或多或少的比其他因素危险.开发者必须在选择开发环境的时候更多地了解特定的应用程序.

rank	▲	☆	vote	url
18	766	373	952	url

---

if __name__ == "__main__": 是干嘛的?

```
# Threading example
import time, thread

def myfunction(string, sleeptime, lock, *args):
    while 1:
        lock.acquire()
        time.sleep(sleeptime)
        lock.release()
        time.sleep(sleeptime)
if __name__ == "__main__":
    lock = thread.allocate_lock()
    thread.start_new_thread(myfunction, ("Thread #: 1", 2, lock))
    thread.start_new_thread(myfunction, ("Thread #: 2", 2, lock))
```

还有 `*args` 在这里是什么意思?

当Python解析器读取一个源文件时,它会执行所有的代码.在执行代码前,会定义一些特殊的变量.例如,如果解析器运行的模块(源文件)作为主程序,它将会把 `__name__` 变量设置成 `"__main__"` .如果只是引入其他的模块, `__name__` 变量将会设置成模块的名字.

假设下面是你的脚本,让我们作为主程序来执行:

```
python threading_example.py
```

当设置完特殊变量,它就会执行 `import` 语句并且加载这些模块.当遇到 `def` 代码段的时候,它就会创建一个函数对象并创建一个名叫 `myfunction` 变量指向函数对象.接下来会读取 `if` 语句并检查 `__name__` 是不是等于 `"__main__"` ,如果是的话他就会执行这个代码段.

这么做的原因是有时你需要你写的模块既可以直接的执行,还可以被当做模块导入到其他模块中去.通过检查是不是主函数,可以让你的代码只在它作为主程序运行时执行,而当其他人调用你的模块中的函数的时候不必执行.

如果想了解更多,请查看这个

rank	▲	☆	vote	url
19	710	261	493	url

理解Python中 `super()` 和 `__init__()` 方法

我试着理解`super()`方法.从表面上看,两个子类实现的功能都一样.我想问它们俩的区别在哪里?

```
class Base(object):
    def __init__(self):
        print "Base created"

class ChildA(Base):
    def __init__(self):
        Base.__init__(self)

class ChildB(Base):
    def __init__(self):
        super(ChildB, self).__init__()

print ChildA(),ChildB()
```

`super()`的好处就是可以避免直接使用父类的名字.但是它主要用于多重继承,这里面有很多好玩的东西.如果还不了解的话可以看看官方文档

注意在Python3.0里语法有所改变:你可以用`super().__init__()`替换`super(ChildB, self).__init__()`.(在我看来非常nice)

rank	▲	☆	vote	url
20	707	342	931	url

`__str__` 和 `repr__` 的区别

`__str__` 和 `repr__` 有什么区别?

首先让我们梳理一下张三的答案:

- 用起来没有什么区别
- `__repr__` 的目的是明确的
- `__str__` 的目的是可读性
- `__str__` 的用法包含 `__repr__`

用法没有什么区别

因为Python的宗旨是易用的,所以这看起来有点不寻常.但是当

rank	▲	☆	vote	url
21	688	153	1320	url

在循环中获取索引(数组下标)

有人知道如何获取列表的索引值吗:

```
ints = [8, 23, 45, 12, 78]
```

当我循环这个列表时如何获得它的索引下标?

如果像C或者PHP那样加入一个状态变量那就太不pythonic了.

最好的选择就是用内建函数enumerate

```
for idx, val in enumerate(ints):
    print idx, val
```

想了解更多可以查看[PEP279].在Python2.x和Python3.x都好使.

rank	▲	✰	vote	url
22	656	133	1124	url

Python中的appen和extend

`append` 和 `extend` 有什么区别?

---

append:

```
x = [1, 2, 3]
x.append([4, 5])
print (x)
```

输出: `[1, 2, 3, [4, 5]]`

extend:

```
x = [1, 2, 3]
x.extend([4, 5])
print (x)
```

输出: `[1, 2, 3, 4, 5]`

rank	▲	☆	vote	url
23	644	117	879	url

字典里添加元素的方法

当一个字典被创建了,能不能在字典里计入一个键?好像没有 `.add()` 的方法.

```
>>> d = {'key':'value'}
>>> print d
{'key': 'value'}
>>> d['mynewkey'] = 'mynewvalue'
>>> print d
{'mynewkey': 'mynewvalue', 'key': 'value'}
```

```
>>> x = {1:2}
>>> print x
{1: 2}

>>> x.update({3:4})
>>> print x
{1: 2, 3: 4}
```

rank	▲	☆	vote	url
24	644	117	879	url

Python中有检查字符串包含的方法吗?

我正在找`string.contains`或者`string.indexof`方法.

我希望:

```
if not somestring.contains("blah"):
   continue
```

你可以用`in`啊:

```
if not "blah" in somestring: continue
```

或者:

```
if "blah" not in somestring: continue
```

rank	▲	☆	vote	url
25	587	87	877	url

在一行里获取多个异常

我知道这样:

```
try:
    # 可能错的地方
except:
    # 如果错了执行这里
```

也知道这样:

```
try:
    # 可能错的地方
except IDontLikeYourFaceException:
    # 给爷笑一个
except YouAreTooShortException:
    # 踩高跷
```

但是我想在两个不同的异常里做同样的事,我能想到的办法:

```
try:
    # 可能错的地方
except IDontLIkeYouException:
    # 滚
except YouAreBeingMeanException:
    # 滚
```

有什么方法能像下面那样:

```
try:
    # 可能错的地方
except IDontLIkeYouException, YouAreBeingMeanException:
    # 滚
```

现在下面的代码根本不好使:

```
try:
    # 可能错的地方
except Exception, e:
    # 滚
```

所以有什么方法可以完成我的要求吗?

用括号扩起来:

```
except (IDontLIkeYouException, YouAreBeingMeanException) as e:
    pass
```

用逗号分割的方法只能在Python2.6和2.7里好使,在Python3中则无效;现在,你应当使用`as`.

rank	▲	☆	vote	url
26	568	149	609	url

类里的静态变量

有可能在python中类有静态变量或方法?用什么语法实现?

变量是在类定义时声明的,不是在类方法或静态变量中:

```
>>> class MyClass:
...     i = 3
...
>>> MyClass.i
3
```

上面的"i"变量是类级别的,所以它是和所有实体级的"i"变量是不一样的,你可以:

```
>>> m = MyClass()
>>> m.i = 4
>>> MyClass.i, m.i
>>> (3, 4)
```

这与C++和Java不一样,但是和C#相同,那就是静态成员不能被实例所引用.

看一下Python教程中关于类和类对象的主题

在这里李四已经回答了静态方法,官方文档内建函数中也提到了.

```
class C:
    @staticmethod
    def f(arg1, arg2, ...): ...
```

rank	▲	☆	vote	url
27	559	101	562	url

如何移除换行符?

这是我用Python编程遇到的最多的问题了,所以我想放到stackoverflow好让我下次Google'chomp python'的时候能得到有用的答案.

试试 `rstrip` 方法:

```
>>> 'test string\n'.rstrip()
'test string'
```

注意Python的 `rstrip` 方法将会默认去掉所有的空白符,而在Perl里只是删除换行符.如果只是删除换行符:

```
>>> 'test string \n'.rstrip('\n')
'test string '
```

同样也有 `lstrip` 和 `strip` 方法:

```
>>> s = " \n  abc   def   "
>>> s.strip()
'abc   def'
>>> s.rstrip()
' \n  abc   def'
>>> s.lstrip()
'abc   def   '
>>>
```

rank	▲	✰	vote	url
28	556	340	568	url

为什么在C++中读取stdin中的行会比Python慢呢?

ps:这个是C++的问题了所以没做翻译.

rank	▲	☆	vote	url
29	569	323	918	url

理解Python切片

对于Python的切片有什么好的资料吗?对我来说理解切片有点难度.它看起来非常有用,但是我还是不能理解它,我正在找有什么好的资料.

非常简单:

```
a[start:end] # 从start开始到end-1结束
a[start:]    # 从start开始直到末尾
a[:end]      # 从头部开始直到end结束
a[:]         # 复制整个列表
```

这里还有一个`step`值,可以用在上面素有例子的后面:

```
a[start:end:step] # 按照step步长直到end-1结束,并不是从start一个个遍历到end
```

重点需要记住的是: `:end`值代表的是不被选中的第一个位置.所以`end`和`start`区别在于选中的元素(如果`step`默认为1)

另一点要说的是`start`或者`end`可能是个负数,也就是从尾部而不是从头部开始计数.所以:

```
a[-1]    # 列表最后一个元素
a[-2:]   # 列表最后两个元素
a[:-2]   # 除了最后两个元素剩下的部分
```

如果你调用的元素多于列表中含有的元素个数,Python也会很友好的表示.例如,如果你请求`a[:-2]`而`a`只含有一个元素,你得到的是一个空列表而不是一个错误信息.有些时候你可能更希望得到这个错误信息,所以你得意识到上面的事可能发生.

rank	▲	☆	vote	url
30	554	230	447	url

怎么在终端里输出颜色?

怎么样用Python在终端里输出带颜色的文本?最好的代替块字符的unicode字符是什么?(What is the best Unicode symbol to represent a solid block?这句话没理解)

这依赖于你用哪种操作系统.最常用的方法就是输出ANSI转义序列.例如,下面的:

```
class bcolors:
    HEADER = '\033[95m'
    OKBLUE = '\033[94m'
    OKGREEN = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    ENDC = '\033[0m'
```

可以这么用上面的代码:

```
print bcolors.WARNING + "Warning: No active frommets remain. Continue?" + bcolors.ENDC
```

这种方法适合OS X,linux和windows系统.还有一些其他的ansi代码可以设置颜色,消除光标或者其他.

如果想要应用更复杂的功能(听起来像是如果你正在写一个游戏),你应该看看"curses"模块,它包含了许多关于次部分复杂操作.Python Curses HowTO可以作为好的介绍.

如果你不使用拓展的ASCII(比如不是一个PC), `#` 或者 `@` 可能是你最好的选择.如果你用的是IBM扩展ascii字符设置,你还可以有更多的选择.176,177,178和219是"块字符"

一些现代的基于文本的程序,像"Swarf Fortress",显示的文本使用图像模式,而用的字体也是传统的计算机字体的图像.你可以在Dwarf Fortress Wiki找到你可以用的字符.

Text Mode Demo Contest 也有许多资料可供参考.

我想可能有点跑题了.我在设计一个史诗级别的基于文本的冒险游戏.祝你好运!

怎么就没人提Python termcolor module.用法相当简单:

```
from termcolor import colored

print colored('hello', 'red'), colored('world', 'green')
```

虽然有点简单,但是对于游戏程序和你想要的"colored blocks"来说足够了.

rank	▲	☆	vote	url
31	549	117	733	url

静态方法

Python有没有静态方法使我可以不用实例化一个类就可以调用,像这样:

```
ClassName.StaticMethod ( )
```

是的,用静态方法装饰器

```
class MyClass(object):
    @staticmethod
    def the_static_method(x):
        print x

MyClass.the_static_method(2) # outputs 2
```

注意有些代码用一个函数而不是 `staticmethod` 装饰器去定义一个静态方法.如果你想支持Python的老版本(2.2和2.3)可以用下面的方法:

```
class MyClass(object):
    def the_static_method(x):
        print x
    the_static_method = staticmethod(the_static_method)

MyClass.the_static_method(2) # outputs 2
```

这个方法和第一个一样,只是没有第一个用装饰器的优雅.

最后,请少用 `staticmethod` 方法!在Python里只有很少的场合适用静态方法,其实许多顶层函数会比静态方法更清晰明了.

文档

rank	▲	☆	vote	url
32	527	250	408	url

为什么用pip比用easy_install的好?

一个推特写道:

> 别用easy_install,除非你想自讨苦吃.那么用pip.

为什么要用 `pip` 而不是 `easy_install` ?如果一个作者上传一个损坏的源文件包(比如丢失文件,没有setup.py),那么 `pip` 和 `easy_install` 都将失效.除了这些,为什么Python使用者(像上面那个)强烈建议用 `pip` 而不是 `easy_install` ?

来自Lan Bicking的自己对于 `pip` 的介绍:

pip是对easy_install以下方面进行了改进:

- 所有的包是在安装之前就下载了.所以不可能出现只安装了一部分.
- 在终端上的输出更加友好.
- 对于动作的原因进行持续的跟踪.例如,如果一个包正在安装,那么pip就会跟踪为什么这个包会被安装.
- 错误信息会非常有用.
- 代码简洁精悍可以很好的编程.
- 不必作为egg存档,能扁平化安装(仍然保存egg元数据)
- 原生的支持其他版本控制系统(Git, Mercurial and Bazaar)
- 卸载包
- 可以简单的定义修改一系列的安装依赖,还可以可靠的赋值一系列包.

rank	▲	☆	vote	url
33	526	73	660	url

把字符串解析成浮点数或者整数

在Python里,怎么样把一个数字型字符串像`"545.2222"`解析成对应的浮点值`545.2222`?或者把一个`"31"`解析成`31`?

我只是想知道怎么样才能解析一个浮点型字符串编程浮点数,或者整型字符串转换成整型?

```
>>> a = "545.2222"
>>> float(a)
545.22220000000004
>>> int(float(a))
545
```

rank	▲	☆	vote	url
34	525	50	764	url

怎么样获取一个列表的长度?

```
items = []
items.append("apple")
items.append("orange")
items.append("banana")

# FAKE METHOD::
items.amount()  # 返回 3
```

怎么样做才对?

---

len函数可以用于Python中许多的类型,包括内建类型和标准库类型.

```
>>len([1,2,3])
3
```

rank	▲	☆	vote	url
36	511	136	405	url

在Python里怎么读取stdin?

我在做code golf挑战,但是所有的问题都需要读取stdin的值.在Python里应当怎么做?

这是我从 Stack Overflow 中学到的:

```
import fileinput

for line in fileinput.input():
    pass
```

`fileinput` 将会迭代在命令行参数给出的文件名的每一行,如果没有给参数,则默认为stdin.

rank	▲	☆	vote	url
37	511	93	416	url

为什么是string.join(list)而不是list.join(string)?

我一直被这个问题困扰.如果这样写更好:

```
my_list = ["Hello", "world"]
print my_list.join("-")
# Produce: "Hello-world"
```

而不是:

```
my_list = ["Hello", "world"]
print "-".join(my_list)
# Produce: "Hello-world"
```

有什么特殊的原因让它不这样做吗?

是因为所有的可迭代对象都能被 `join` ,不仅仅是列表,但结果是我们一般要 `join` 的都是字符串.

例如:

```
import urllib2
print '\n############\n'.join(urllib2.urlopen('http://data.stackexchange.com/users/7095'))
```

rank	▲	☆	vote	url
38	508	92	515	url

在Python里获取当前时间

用什么方法或者模块获取当前时间?

```
>>> import datetime
>>> datetime.datetime.now()
datetime(2009, 1, 6, 15, 8, 24, 78915)
```

如果只要时间:

```
>>> datetime.datetime.time(datetime.datetime.now())
datetime.time(15, 8, 24, 78915)
```

同样功能但是更紧凑的写法:

```
>>> datetime.datetime.now().time()
```

获取更多信息,看文档.

如果还要节省输出,可以在从 `datetime` 模块import `datetime` 对象:

```
>>> from datetime import datetime
```

然后把所有 `datetime.` 从头部去掉.

rank	▲	☆	vote	url
39	502	134	633	url

在Python中列出目录中的所有文件

怎么样用Python列出一个目录的所有文件并且存进一个列表?

`os.listdir()` 可以获得一个目录中所有文件或者子目录.

如果你只想要文件的话,你也可以用 `os.path` 把其他的过滤掉:

```
from os import listdir
from os.path import isfile, join
onlyfiles = [ f for f in listdir(mypath) if isfile(join(mypath,f)) ]
```

或者你可以用 `os.walk()` ,它遍历每个目录将会返回两个列表(一个文件列表,一个目录列表),如果你想要顶层目录只需要在第一次迭代后break一下即可.

```
from os import walk

f = []
for (dirpath, dirnames, filenames) in walk(mypath):
    f.extend(filenames)
    break
```

最后,如果你想增加列表可以像上面那样用 `.extend()` 或者:

```
>>> q = [1,2,3]
>>> w = [4,5,6]
>>> q = q + w
>>> q
[1,2,3,4,5,6]
```

我个人更喜欢 `.extend()`

rank	▲	☆	vote	url
40	495	206	572	url

在Python怎么样才能把列表分割成同样大小的块?

我有一个任意长度的列表,我需要把它们切成相同大小并且使用它们.有一些简单的方法,比如设置一个计数器和两个列表,当第二个列表装满后,把它存进第一个列表然后让第二个列表变空一遍存放下一轮的数据,但是这么做代价太大了.

我想是不是还有其他更好的方法,比如生成器

像下面这样:

```
l = range(1, 1000)
print chunks(l, 10) -> [ [ 1..10 ], [ 11..20 ], .., [ 991..999 ] ]
```

我试着在 `itertools` 找一些有用的方法,但是还没有找到.也可能是我还没有看到.

相似问题:What is the most “pythonic” way to iterate over a list in chunks?

下面是生成器方法:

```
def chunks(l, n):
    """ Yield successive n-sized chunks from l.
    """
    for i in xrange(0, len(l), n):
        yield l[i:i+n]
```

```
import pprint
pprint.pprint(list(chunks(range(10, 75), 10)))
[[10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
 [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
 [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
 [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
 [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
 [70, 71, 72, 73, 74]]
```

一句话

```
tuple(l[i:i+n] for i in xrange(0, len(l), n))
```

rank	▲	☆	vote	url
41	487	108	705	url

检查一个字符串是否是一个数字

如果一个字符串可以被看做一个数字那么有什么好的方法可以检测出来?

我能想到的方法:

```
def is_number(s):
    try:
        float(s)
        return True
    except ValueError:
        return False
```

我还没有想到什么更好的方法.

对字符串对象用 `isdigit()` 方法:

```
>>> a = "03523"
>>> a.isdigit()
True
```

```
>>> b = "963spam"
>>> b.isdigit()
False
```

String Methods - isdigit()

同样也有unicode的方法，但是我不太熟悉Unicode - Is decimal/decimal

rank	▲	☆	vote	url
42	465	90	791	url

查找列表中某个元素的下标

比如`["foo","bar","baz"]`和它的一个元素`"bar"`,用什么方法能找到它的下标(1)?

```
>>> ["foo","bar","baz"].index('bar')
1
```

参考资料:Data Structures > More on Lists

rank	▲	✰	vote	url
43	465	86	790	url

获得一个字符串的子串

有什么方法获得一个字符串的字串,比如从一个字符串的第三个字符到最后.

可能是 `myString[2:end]`?

```
>>> x = "Hello World!"
>>> x[2:]
'llo World!'
>>> x[:2]
'He'
>>> x[:-2]
'Hello Worl'
>>> x[-2:]
'd!'
>>> x[2:-2]
'llo Worl'
```

上面的概念在Python中叫"slicing"(切片),不止在字符串上有这个方法.在这里有详细的解释.

rank	▲	☆	vote	url
44	455	162	534	url

为什么代码在一个函数里运行的更快?

```
def main():
    for i in xrange(10**8):
        pass
main()
```

在Python中运行速度:

```
real    0m1.841s
user    0m1.828s
sys     0m0.012s
```

然而不把它放在函数里:

```
for i in xrange(10**8):
    pass
```

它比上一个运行时间更长

```
real    0m4.543s
user    0m4.524s
sys     0m0.012s
```

为啥会这样?

注:这个计时是用LinuxBASH中的时间函数.

你或许想问为什么存取一个本地变量比全局变量要快.这有关于CPython的实现细节.

记住CPython解析器运行的是被编译过的字节码(bytecode).当一个函数被编译后,局部变量被存储在了固定大小的数组(不是一个 `dict`),而变量名赋值给了索引.这就是为什么你不能动态的为一个函数添加局部变量.检查一个局部变量就好像是一个指针去查找列表,对于在 `PyObject` 上的引用计数的增长是微不足道的.

相反的在查找全局变量(`LOAD_GLOBAL`)时,涉及到的是一个实实在在的 `dict` 的哈希查找.顺便说一句,这就是为什么当你想要一个全局变量你必须要在前面加上 `global i` :如果你在一个区域内指定一个变量,编译器就会建立一个 `STORE_FAST` 的入口,除非你不让它那么做.

在说一句,全局查找速度也不慢.真正拖慢速度的是像 `foo.bar` 这样的属性查找!

在一个函数里,字节码是这样的:

```
2           0 SETUP_LOOP              20 (to 23)
            3 LOAD_GLOBAL              0 (xrange)
            6 LOAD_CONST               3 (100000000)
            9 CALL_FUNCTION            1
           12 GET_ITER
      >>   13 FOR_ITER                 6 (to 22)
           16 STORE_FAST               0 (i)

3          19 JUMP_ABSOLUTE           13
      >>   22 POP_BLOCK
      >>   23 LOAD_CONST               0 (None)
           26 RETURN_VALUE
```

如果在全局,字节码:

```
1           0 SETUP_LOOP              20 (to 23)
            3 LOAD_NAME                0 (xrange)
            6 LOAD_CONST               3 (100000000)
            9 CALL_FUNCTION            1
           12 GET_ITER
      >>   13 FOR_ITER                 6 (to 22)
           16 STORE_NAME               1 (i)

2          19 JUMP_ABSOLUTE           13
      >>   22 POP_BLOCK
      >>   23 LOAD_CONST               2 (None)
           26 RETURN_VALUE
```

两者的区别是 `STORE_FAST` 比 `STORE_NAME` 要快很多.这是因为在函数里 `i` 是一个局部变量而在全局区域它是一个全局变量.

可以看一下字节编码,用到了 `dis` 模块.我可以直接解析函数,但是要解析全局代码必须用 `compile` 内建模块.

rank	▲	✰	vote	url
45	441	233	760	url

合并列表中的列表

可能重复的问题:

- Flattening a shallow list in Python
- Comprehension for flattening a sequence of sequences?

我想是不是有更好的方法

我可以用一个循环来做,但是除了这样做还有什么更cool的用一行来做的方法?我用 `reduce` 来做,但是我得到了个错误.

代码:

```
l = [[1,2,3],[4,5,6], [7], [8,9]]
reduce(lambda x,y: x.extend(y),l)
```

错误信息:

```
Traceback (most recent call last): File "", line 1, in File "", line 1, in AttributeError: 'NoneType' object has r
```

`[item for sublist in l for item in sublist]` 到目前为止来看是最快的方法.

可以用 `timeit` 模块验证一下:

```
$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' '[item for sublist in l for item in sublist]'
10000 loops, best of 3: 143 usec per loop
$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' 'sum(l, [])'
1000 loops, best of 3: 969 usec per loop
$ python -mtimeit -s'l=[[1,2,3],[4,5,6], [7], [8,9]]*99' 'reduce(lambda x,y: x+y,l)'
1000 loops, best of 3: 1.1 msec per loop
```

解释:

当有L个子串的时候用 `+` (即 `sum`)的时间复杂度是 `O(L**2)` --每次迭代的时候作为中间结果的列表的长度就会越来越长,而且前一个中间结果的所有项都会再拷贝一遍给下一个中间结果.所以当你的列表l含有L个字串:l列表的第一项需要拷贝L-1次,而第二项要拷贝L-2次,以此类推;所以总数为 `I * (L**2)/2` .

列表推导式(list comprehension)只是生成一个列表,每次运行只拷贝一次(从开始的地方拷贝到最终结果).

rank	▲	✰	vote	url
46	1033	1642	377	url

Python使用什么IDE?

结果:

```
                                   Rapid Application Development -.
                                       Integrated DB Support -+   |
                                            GUI Designer  -+  |   |
                                         Unit Testing -+   |  |   |
                                    Code Templates -.  |   |  |   |
                                   Code Folding -+  |  |   |  |   |
                      UML Editing / Viewing -+  |  |  |   |  |   |
                         Line Numbering -+   |  |  |  |   |  |   |
                    Bracket Matching -+  |   |  |  |  |   |  |   |
                     Smart Indent -+  |  |   |  |  |  |   |  |   |
        Source Control Integration -+  |  |  |   |  |  |  |   |  |   |
                  Error Markup  -+  |  |  |  |   |  |  |  |   |  |   |
 Integrated Python Debugging -+  |  |  |  |  |   |  |  |  |   |  |   |
   Multi-Language Support -+  |  |  |  |  |  |   |  |  |  |   |  |   |
 Auto Code Completion -+   |  |  |  |  |  |  |   |  |  |  |   |  |   |
Commercial / Free --+  |   |  |  |  |  |  |  |   |  |  |  |   |  |   |
Cross Platform -+   |  |   |  |  |  |  |  |  |   |  |  |  |   |  |   |
               _|___|__|___|__|__|__|__|__|__|___|__|__|__|___|__|___|_
               |CP|C/F|AC|MLS|PD|EM|SC|SI|BM|LN|UML|CF|CT|UT|UID|DB|RAD|comments
               +--+---+--+---+--+--+--+--+--+--+---+--+--+--+---+--+---+
BlackAdder     |Y | C |  |   |  |  |  |Y |  |  |   |Y |  |  |   |  |   |
BlueFish       |L |   |  |   |  |  |  |  |  |  |   |  |  |  |   |  |   |
Boa Constructor|Y | F |Y |   |Y |Y |  |Y |Y |Y | Y |Y |Y |  |   |  |   |
ConTEXT        |W | C |  |   |  |  |  |  |  |  |   |  |  |  |   |  |   |
DABO           |Y |   |  |   |  |  |  |  |  |  |   |  |  |  |   |  |   |
DreamPie       |  | F |  |   |  |  |  |  |  |  |   |  |  |  |   |  |   |
Dr.Python      |  | F |  |   |  |Y |  |  |  |  |   |  |  |  |   |  |   |
Editra         |Y | F |Y | Y |  |  |Y |Y |Y |Y |   |Y |  |  |   |  |   |
Emacs          |Y | F |Y | Y |Y |Y |Y |Y |Y |Y | Y |Y |Y |Y |   |  |   |
Eric Ide       |Y | F |Y |   |Y |Y |  |Y |  |Y |   |Y |  |Y |   |  |   |
E-Texteditor   |W |   |  |   |  |  |  |  |  |  |   |  |  |  |   |  |   |
Geany          |Y | F |Y*| Y |  |  |  |Y |Y |Y |   |Y |  |  |   |  |   |* very limited
Gedit          |Y | F |Y¹| Y |  |  |  |Y |Y |Y |   |  |Y²|  |   |  |   |¹ with plugin ² sort of
Idle           |Y | F |Y |   |  |  |  |  |  |  |   |  |  |  |   |  |   |
JEdit          |Y | F |  | Y |  |  |  |  |Y |Y |   |Y |  |  |   |  |   |
KDevelop       |Y | F |  | Y |  |  |Y |Y |Y |Y |   |Y |  |  |   |  |   |
Komodo         |Y |C/F|Y | Y |Y |Y |Y |Y |Y |Y |   |Y |Y |Y |   |Y |   |
NetBeans       |Y | F |Y | Y |Y |  |Y |Y |Y |Y | Y |Y |Y |Y |   |  | Y |
NotePad++      |W | F |  | Y |  |  |  |  |  |Y |   |  |  |  |   |  |   |
Pfaide         |W | C |Y | Y |  |  |  |Y |Y |Y |   |Y |Y |  |   |  |   |
PIDA           |LW| F |Y | Y |  |  |  |Y |Y |Y |   |Y |  |  |   |  |   |VIM based
PTVS           |W | F |Y | Y |Y |Y |Y |Y |Y |Y |   |Y |  |  |Y* |  | Y |*WPF bsed
PyCharm        |Y | C |Y | Y*|Y |  |Y |Y |Y |Y |   |Y |  |Y |   |  |   |* javascript
PyDev(Eclipse) |Y | F |Y | Y |Y |Y |Y |Y |Y |Y | Y |Y |Y |Y |   |  |   |
Pyscripter     |W | F |Y |   |Y |Y |  |Y |  |Y |   |  |Y |Y |   |  |   |
PythonWin      |W | F |Y |   |Y |  |  |Y |Y |  |   |Y |  |  |   |  |   |
SciTE          |Y | F |  | Y |  |Y |  |  |Y |Y |   |Y |Y |  |   |  |   |
ScriptDev      |W | C |Y | Y |Y |Y |  |Y |Y |Y |   |Y |Y |  |   |  |   |
SPE            |  | F |Y |   |  |  |  |  |  |  |Y  |  |  |  |   |  |   |
Spyder         |Y | F |Y |   |Y |Y |  |Y |Y |Y |   |  |  |  |   |  |   |
Sublime Text   |Y | C |Y | Y |  |  |  |Y |Y |Y |   |  |Y |  |   |  |   |extensible w/python
TextMate       |M |   |  | Y |  |  |  |Y |Y |Y |   |Y |Y |  |   |  |   |
UliPad         |Y | F |Y | Y |Y |  |  |Y |Y |  |   |  |Y |Y |   |  |   |
Vim            |Y | F |Y | Y |Y |Y |Y |Y |Y |Y |   |Y |Y |Y |   |  |   |
WingIde        |Y | C |Y | Y*|Y |Y |Y |Y |Y |Y |   |Y |Y |Y |   |  |   |* support for C
Zeus           |W | C |  |   |  |  |Y |Y |Y |Y |   |Y |Y |  |   |  |   |
               +--+---+--+---+--+--+--+--+--+--+---+--+--+--+---+--+---+
               |CP|C/F|AC|MLS|PD|EM|SC|SI|BM|LN|UML|CF|CT|UT|UID|DB|RAD|
               |__|___|__|___|__|__|__|__|__|__|___|__|__|__|___|__|___|
```

缩写:

- CP - Cross Platform
- C - Commercial
- F - Free
- AC - Automatic Code-completion
- MLS - Multi-Language Support
- PD - Integrated Python Debugging
- EM - ErrorMarkup
- SC - Source Control integration
- SI - Smart Indent
- BM - Bracket Matching
- LN - Line Numbering
- UML - UML editing / viewing
- CF - Code Folding
- CT - Code Templates
- UT - Unit Testing
- UID - GUI Designer (for example, Qt, Eric, ..)
- DB - integrated database support
- RAD - Rapid application development support
- L - Linux
- W - Windows

这里我没有提最基本的语法高亮（默认都带）

我只是总结一下.

rank	▲	☆	vote	url
47	435	86	477	url

检查一个键在字典中是否存在

在更新字典之前想检查键是否存在.我写了如下代码:

```
if 'key1' in dict.keys():
  print "blah"
else:
  print "boo"
```

我想这不是最好的方法,还有什么更好的方法?

用 `in` .

```
d = dict()

for i in xrange(100):
    key = i % 10
    if key in d:
        d[key] += 1
    else:
        d[key] = 1
```

如果你想要一个默认值,你可以用 `dict.get()` :

```
d = dict()

for i in xrange(100):
    key = i % 10
    d[key] = d.get(key, 0) + 1
```

如果相对所有值设置默认值可以用 `collections` 模块的 `defaultdict` 函数:

```
from collections import defaultdict

d = defaultdict(lambda: 0)

for i in xrange(100):
    d[i % 10] += 1
```

但是总而言之 `in` 是最好的方法.

rank	▲	☆	vote	url
48	432	75	793	url

在列表中随机取一个元素

例如我有如下列表:

```
foo = ['a', 'b', 'c', 'd', 'e']
```

从列表中随机取一个元素最好的方法是什么?

```
import random

foo = ['a', 'b', 'c', 'd', 'e']
print(random.choice(foo))
```

rank	▲	☆	vote	url
49	432	198	616	url

通过列表中字典的值对列表进行排序

我的到了一个字典的列表,我想对字典的值进行排序.

```
[{'name':'Homer', 'age':39}, {'name':'Bart', 'age':10}]
```

对name进行排序,应当是:

```
[{'name':'Bart', 'age':10}, {'name':'Homer', 'age':39}]
```

用key比用cmp更清晰明了:

```
newlist = sorted(list_to_be_sorted, key=lambda k: k['name'])
```

或者其他人的建议:

```
from operator import itemgetter
newlist = sorted(list_to_be_sorted, key=itemgetter('name'))
```

rank	▲	☆	vote	url
50	404	65	510	url

复制文件

怎么在Python里赋值文件?在 `os` 下没找到复制的方法.

`shutil` 有许多的方法.其中之一就是:

```
copyfile(src, dst)
```

把 `src` 文件的内容复制给 `dst` .目的地址必须是可写的;否则将会出现 `IOError` 错误.如果 `dst` 已经存在,将会被覆盖.一些像字符或者块设备不能用这个方法赋值. `src` 和 `dst` 是路径名的字符串形式.

rank	▲	☆	vote	url
51	403	213	451	url

---

error: Unable to find vcvarsall.bat

我试着安装Python的`dulwich`包:

```
pip install dulwich
```

但是得到了个错误:

```
error: Unable to find vcvarsall.bat
```

如果手动安装也会出现相同的错误:

```
> python setup.py install
running build_ext
building 'dulwich._objects' extension
error: Unable to find vcvarsall.bat
```

对于Windows安装:

当安装包运行`setup.py`时,Python2.7会先寻找已经安装的Visual Studio 2008.你可以在运行`setup.py`直线设置正确的`VS90COMNTOOLS`变量.

根据Visual Studio的不同执行不同的命令:

- Visual Studio 2010 (VS10): SET `VS90COMNTOOLS=%VS100COMNTOOLS%`
- Visual Studio 2012 (VS11): SET `VS90COMNTOOLS=%VS110COMNTOOLS%`
- Visual Studio 2013 (VS12): SET `VS90COMNTOOLS=%VS120COMNTOOLS%`

rank	▲	☆	vote	url
52	403	113	370	url

如何移除用easy_install下载的包?

用 easy_install 下载包非常方便.但是我至今没发现怎么移除下载的包.

有什么更好的方法来移除包?如果我手动移除的话需要修改什么文件?(比如 rm /usr/local/lib/python2.6/dist-packages/my_installed_pkg.egg)

---

pip,另一个选择,提供了"uninstall"命令.

按着说明安装pip:

```
$ wget https://bootstrap.pypa.io/get-pip.py
$ python get-pip.py
```

然后就可以用 pip uninstall 去移除用 easy_install 安装的包.

rank	▲	☆	vote	url
53	400	168	184	url

从相对路径引入一个模块

怎么用相对路径引入一个模块?

例如,如果 `dirFoo` 包含 `Foo.py` 和 `dirBar` ,而 `dirBar` 包含 `Bar.py` ,怎么样才能在 `Foo.py` 里引入 `Bar.py` ?

结构:

```
dirFoo\
    Foo.py
    dirBar\
        Bar.py
```

`Foo` 希望包含 `Bar` ,但是重新组织文件结构好像不太好.

确定 `djirBar` 有一个 `__init__.py` 文件--使你的目录包含一个Python包.

rank	▲	☆	vote	url
54	397	83	538	url

如何知道一个对象有一个特定的属性?

有什么方法可以检测一个对象是否有某些属性?比如:

```
>>> a = SomeClass()
>>> a.someProperty = value
>>> a.property
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: SomeClass instance has no attribute 'property'
```

怎样在使用之前知道 `a` 是否有 `property` 这个属性?

试试 `hasattr()` :

```
if hasattr(a, 'property'):
    a.property
```

在大多数实际情况下,如果一个属性有很大可能存在,那么就直接调用它或者让它引发异常,或者用 `try/except` 捕获.这种方法比 `hasattr` 快.如果这个属性很多情况下不在,或者你不确定,那么用 `hasattr` 将会比触法异常更快.

rank	▲	☆	vote	url
55	392	282	480	url

---

*args 和 **kwargs

我对 *args 和 **kwargs 的概念有点不理解.

到目前位置我了解到:

- *args = 参数列表-作为可选参数
- **kwargs = 字典-参数的关键字作为键,它们的值作为字典的值.

??

说实话我不裂解这对编程来说有什么用?(我知道它,但是对他不了解.)

或许:

我想把字典或者列表当做参数的同时也把它们当成通配符,所以可以传递任何参数?

有什么实际例子来解释怎么用 *args 和 **kwargs ?

通过教程我只是用了"*"和一个变量名.

还是 *args 和 **kwargs 只是一个占位符,你们在代码里用 *args 和 **kwargs 吗?

---

* 和 ** 的语法.用 *args 和 **kwargs 只是为了方便并没有强制使用它们.

当你不确定你的函数里将要传递多少参数时你可以用 *args .例如,它可以传递任意数量的参数:

```
>>> def print_everything(*args):
        for count, thing in enumerate(args):
...         print '{0}. {1}'.format(count, thing)
...
>>> print_everything('apple', 'banana', 'cabbage')
0. apple
1. banana
2. cabbage
```

相似的, **kwargs 允许你使用没有事先定义的参数名:

```
>>> def table_things(**kwargs):
...     for name, value in kwargs.items():
...         print '{0} = {1}'.format(name, value)
...
>>> table_things(apple = 'fruit', cabbage = 'vegetable')
cabbage = vegetable
apple = fruit
```

你也可以混着用.命名参数首先获得参数值然后所有的其他参数都传递给 *args 和 **kwargs .命名参数在列

表的最前端.例如:

```
def table_things(titlestring, **kwargs)
```

`*args` 和 `**kwargs` 可以同时在函数的定义中,但是 `*args` 必须在 `**kwargs` 前面.

当调用函数时你也可以用 `*` 和 `**` 语法.例如:

```
>>> def print_three_things(a, b, c):
...     print 'a = {0}, b = {1}, c = {2}'.format(a,b,c)
...
>>> mylist = ['aardvark', 'baboon', 'cat']
>>> print_three_things(*mylist)
a = aardvark, b = baboon, c = cat
```

就像你看到的一样,它可以传递列表(或者元组)的每一项并把它们解包.注意必须与它们在函数里的参数相吻合.当然,你也可以在函数定义或者函数调用时用 `*`.

rank	▲	☆	vote	url
56	390	59	554	url

如何获取实例的类名

如何才能获取一个实例对象的类名?

我想或许the inspect module可以实现,但是好像还没有发现什么能帮我的.或许可以分析 `__class__` 成员,但是我不知道怎么用.

你试没试类里的 `__name__` 属性?比如 `x.__class__.__name__` 或许可以得到你想要的

```
>>> import itertools
>>> x = itertools.count(0)
>>> x.__class__.__name__
'count'
```

注:如果你用的是新式类(从Python2.2),你可以调用 `type()` 函数来替代:

```
>>> type(x).__name__
'count'
```

像 `int` 这种内建方法也可以:

```
>>> (5).__class__.__name__
'int'
>>> type(5).__name__
'int'
```

rank	▲	☆	vote	url
57	386	107	665	url

字典推导式

我喜欢列表推导式的语法.

它能不能用来创建字典?这样:

```
mydict = {(k,v) for (k,v) in blah blah blah}  # doesn't work :(
```

在Python2.6或更早的版本,字典生成器可以接受迭代的键/值对:

```
d = dict((key, value) for (key, value) in iterable)
```

从Python2.7或者3以后,你可以直接用字典推导式语法:

```
d = {key: value for (key, value) in iterable}
```

当然,你可以用任何方式的迭代器(元组,列表,生成器..),只要可迭代对象的元素中有两个值.

```
d = {value: foo(value) for value in sequence if bar(value)}

def key_value_gen(k):
   yield chr(k+65)
   yield chr((k+13)%26+65)
d = dict(map(key_value_gen, range(26)))
```

rank	▲	☆	vote	url
58	383	111	805	url

反转字符串

在Python里 `str` 没有内建的 `reverse` 函数.实现字符串反转最好的方法是什么?

如果答案很简单的话,那么最有效的是什么.是不是 `str` 要转换成一个不同的对象.

这个怎么样:

```
>>> 'hello world'[::-1]
'dlrow olleh'
```

这是拓展的切片语法.用 `[begin:end:step]` 实现-从begin开始到end结束步长是-1的话就可以反转一个字符串.

rank	▲	✰	vote	url
59	383	111	805	url

通过函数名的字符串来调用这个函数

加入我们有个模块叫 `foo` ,而我有一个 `"bar"` 字符串.调用 `foo.bar()` 有什么最好的方法?

我需要返回函数值,为什么我不能用 `eval` .我想应该能用 `eval` 来定义一个函数来返回调用的结果,但是我希望更优雅的方法.

假设 `foo` 有一个 `bar` 方法:

```
import foo
methodToCall = getattr(foo, 'bar')
result = methodToCall()
```

第二行和第三行可以简写:

```
result = getattr(foo, 'bar')()
```

如果让上面的代码更有意义. `getattr` 可以用在实例绑定,模块级方法,类方法...等等

rank	▲	✰	vote	url
60	383	111	805	url

如何测量脚本运行时间?

我在Project Euler发现好多这样的问题,许多其他地方也问怎么测量执行时间.但是有的时候答案有点kludgey-比如,在`__main__`中加入时间代码,所以我想在这里分享一下解决方案.

Python自带了一个叫`cProfile`的分析器.它不仅实现了计算整个时间,而且单独计算每个函数运行时间,并且告诉你这个函数被调用多少次,它可以很容易的确定你要优化的值.

你可以在你的代码里或是交互程序里调用,像下面这样:

```
import cProfile
cProfile.run('foo()')
```

更有用的是,你可以在运行脚本的时候用`cProfile`:

```
python -m cProfile myscript.py
```

为了使用更简单,我做了一个小脚本名字叫`profile.bat`:

```
python -m cProfile %1
```

所以我可以这么调用了:

```
profile euler048.py
```

下面是结果:

```
1007 function calls in 0.061 CPU seconds

Ordered by: standard name
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    1    0.000    0.000    0.061    0.061 <string>:1(<module>)
 1000    0.051    0.000    0.051    0.000 euler048.py:2(<lambda>)
    1    0.005    0.005    0.061    0.061 euler048.py:2(<module>)
    1    0.000    0.000    0.061    0.061 {execfile}
    1    0.002    0.002    0.053    0.053 {map}
    1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler objects}
    1    0.000    0.000    0.000    0.000 {range}
    1    0.003    0.003    0.003    0.003 {sum}
```

注:更新一个来自PyCon2013的视频网址: http://lanyrd.com/2013/pycon/scdywg/

rank	▲	☆	vote	url
61	374	45	700	url

获取列表最后一个元素

在Python里,如何获取一个列表的最后一个元素?

`some_list[-1]` 最短最Pythonic的方法.

事实上你可以用这个语法做好多事. `some_list[-n]` 语法获取倒数第n个元素.所以 `some_list[-1]` 获取最后一个元素, `some_list[-2]` 获取倒数第二个,等等.最后 `some_list[-len(some_list)]` 可以获取第一个元素~~

你也可以用这种方法获取列表元素,例如:

```
>>> some_list = [1, 2, 3]
>>> some_list[-1] = 5 # Set the last element
>>> some_list[-2] = 3 # Set the second to last element
>>> some_list
[1, 3, 5]
```

rank	▲	☆	vote	url
62	370	162	300	url

Python中用什么代替switch语句?

我想写一个函数,实现输入一个值对应输出另一个值.

在其他语言我可以用 `switch` 或者 `case` 语句,但是Python里没有 `switch` .Python里如何解决这个问题?

你可以用一个字典:

```
def f(x):
    return {
        'a': 1,
        'b': 2,
    }[x]
```

如果你希望设置一个默认值可以用字典的 `get` 方法:

```
def f(x):
    return {
        'a': 1,
        'b': 2,
        }.get(x, 9)    # 9 is default if x not found
```

rank	▲	☆	vote	url
63	367	163	862	url

生成包含大写字母和数字的随机字符串

我希望生成N大小的字符串.

里面只含有数字和大写字母,比如:

- 6U1S75
- 4Z4UKK
- U911K4

有没有什么Pythonic的方法?

一行写的答案:

```
''.join(random.choice(string.ascii_uppercase + string.digits) for _ in range(N))
```

为了重用可以这样写:

```
>>> import string
>>> import random
>>> def id_generator(size=6, chars=string.ascii_uppercase + string.digits):
...    return ''.join(random.choice(chars) for _ in range(size))
...
>>> id_generator()
'G5G74W'
>>> id_generator(3, "6793YUIO")
'Y3U'
```

如何工作?

我们引入 `string` 和 `random` 模块, `string` 模块包含了所有常用的ASCII字符, `random` 模块处理随机数的生成.

`string.ascii_uppercase + string.digits` 是一个包含大写字母和数字的列表:

```
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
>>> string.ascii_uppercase + string.digits
'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
```

然后我们用列表推导式创建 `n` 个元素的列表:

```
>>> range(4) # n个数字的列白哦
[0, 1, 2, 3]
>>> ['elem' for _ in range(4)] # 在一个列表创建4次'elem'
['elem', 'elem', 'elem', 'elem']
```

在上面的例子中,我们用 `[]` 来创建列表,和 `id_generator` 函数不一样,生成器不再内存中创建列表,只是在程序运行时一个一个的生成列表元素(更多见这里)

下面是在列表中随机取出一个元素:

```
>>> random.choice("abcde")
'a'
>>> random.choice("abcde")
'd'
>>> random.choice("abcde")
'b'
```

所以 `random.choice(chars) for _ in range(size)` 会取出 `size` 个字符.字符随机的在 `chars` 中取出:

```
>>> [random.choice('abcde') for _ in range(3)]
['a', 'b', 'b']
>>> [random.choice('abcde') for _ in range(3)]
['e', 'b', 'e']
>>> [random.choice('abcde') for _ in range(3)]
['d', 'a', 'c']
```

然后我们可以用空字符串把它们连起来成为一个字符串:

```
>>> ''.join(['a', 'b', 'b'])
'abb'
>>> [random.choice('abcde') for _ in range(3)]
['d', 'c', 'b']
>>> ''.join(random.choice('abcde') for _ in range(3))
'dac'
```

rank	▲	☆	vote	url
64	365	174	322	url

在实例名字前单下划线和双下划线的含义

我想刨根问底,这到底是什么意思?解释一下他俩的区别.

单下划线

在一个类中的方法或属性用单下划线开头就是告诉别的程序这个属性或方法是私有的.然而对于这个名字来说并没有什么特别的.

引自PEP-8:

> 单下划线:"内部使用"的弱指示器.比如,from M import * 将不会引进用但下划线开头的对象.

双下划线

来自Python文档:

> 任何 `__spam` 形式(至少两个下划线开头,至多一个下划线结尾)都是代替 `_classname__spam` ,其中classname是当前类的名字.This mangling is done without regard to the syntactic position of the identifier.所以它能用来定义私有类的实例和类变量,方法,在全局中的变量,甚至是实例中的变量.可以区别不同类的实例.

例子

```
>>> class MyClass():
...     def __init__(self):
...             self.__superprivate = "Hello"
...             self._semiprivate = ", world!"
...
>>> mc = MyClass()
>>> print mc.__superprivate
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: myClass instance has no attribute '__superprivate'
>>> print mc._semiprivate
, world!
>>> print mc.__dict__
{'_MyClass__superprivate': 'Hello', '_semiprivate': ', world!'}
```

`__foo__` :一种约定,Python内部的名字,用来区别其他用户自定义的命名,以防冲突.

`_foo` :一种约定,用来指定变量私有.程序员用来指定私有变量的一种方式.

`__foo` :这个有真正的意义:解析器用 `_classname__foo` 来代替这个名字,以区别和其他类相同的命名.

在Python中没有其他形式的下划线了.

这种约定方式和类,变量,全局变量等没有区别.

rank	▲	☆	vote	url
65	357	50	683	url

合并两个列表

怎样合并两个列表?

例如:

```
listone = [1,2,3]
listtwo = [4,5,6]
```

我期待:

```
mergedlist == [1, 2, 3, 4, 5, 6]
```

在Python中非常容易.

```
mergedlist = listone + listtwo
```

rank	▲	☆	vote	url
66	357	52	148	url

输出到stderr

我知道至少三种方法这么做:

```
import sys
 sys.stderr.write('spam\n')

 print >> sys.stderr, 'spam'

 from __future__ import print_function
 print('spam', file=sys.stderr)
```

但是看起来和python的第13条宗旨相违背,所以有什么更好的方法吗?或者上面那些方法有什么可以改进?

> There should be one — and preferably only one — obvious way to do it.

我发现只有这个方法短小+便捷+可读性好:

```
from __future__ import print_function
def warning(*objs):
    print("WARNING: ", *objs, file=sys.stderr)
```

我会选择`sys.stderr.write()`,更可读而且言简意赅的突出重点,最主要的是在所有版本都可以用.

注:`Pythonic`是除了可读和效率意外才考虑的事情...如果记住前两条的话80%的代码就已经是`Pythonic`了. 列表推导式是一个'big thing'所以不经常用(可读性).

rank	▲	☆	vote	url
67	356	78	226	url

把字符串转化成时间

短小精悍.我有一个特别大的列表存储了下面的字符串:

```
Jun 1 2005  1:33PM
Aug 28 1999 12:00AM
```

我想把它们转化成合适的时间格式存进数据库,所以我需要把它们变成真正的时间对象.

非常感谢帮助.

检查time模块的strptime函数.它是strftime的转换.

```
from datetime import datetime

date_object = datetime.strptime('Jun 1 2005  1:33PM', '%b %d %Y %I:%M%p')
```

rank	▲	☆	vote	url
68	354	80	465	url

查看一个对象的类型

有什么方便的方法查看一个对象的类型?

为了获得对象的类型,可以用内建函数 type() .把对象作为唯一的参数传递将会返回这个对象的类型:

```
>>> type([]) is list
True
>>> type({}) is dict
True
>>> type('') is str
True
>>> type(0) is int
True
```

当然也对自定义类型也有用:

```
>>> class Test1 (object):
        pass
>>> class Test2 (Test1):
        pass
>>> a = Test1()
>>> b = Test2()
>>> type(a) is Test1
True
>>> type(b) is Test2
True
```

注意 type() 只会返回对象的直接类型,不会告诉你继承类型.

```
>>> type(b) is Test1
False
```

可以用 isinstance 函数.也对内建函数管用:

```
>>> isinstance(b, Test1)
True
>>> isinstance(b, Test2)
True
>>> isinstance(a, Test1)
True
>>> isinstance(a, Test2)
False
>>> isinstance([], list)
True
>>> isinstance({}, dict)
True
```

`isinstance()` 通常是确定一个对象类型更好的方法,因为它接受派生类型.所以除非你确实需要知道对象的类型(一些其他原因),用 `isinstance()` 比 `type()` 更好.

`isinstance()` 的第二个参数也接受类型的元组,所以也可以一次检查多种类型.如果是这些类型里的 `isinstance()` 将会返回true:

```
>>> isinstance([], (tuple, list, set))
True
```

rank	▲	☆	vote	url
69	352	48	520	url

手动抛出异常

我想故意制造一个错误,所以我可以转到`excepy:`语句

我怎么做?

不能在Pythonic了;

```
raise Exception("I know python!")
```

想得到更多信息,看这里

rank	▲	☆	vote	url
70	354	145	282	url

字符串格式化:%和.format

Python2.6推出了[str.format()]方法,和原有的%格式化方式有小小的区别.那个方法更好?

1. 下面的方法有同样的输出,它们的区别是什么?

    ```
    #!/usr/bin/python
    sub1 = "python string!"
    sub2 = "an arg"

    a = "i am a %s" % sub1
    b = "i am a {0}".format(sub1)

    c = "with %(kwarg)s!" % {'kwarg':sub2}
    d = "with {kwarg}!".format(kwarg=sub2)

    print a    # "i am a python string!"
    print b    # "i am a python string!"
    print c    # "with an arg!"
    print d    # "with an arg!"
    ```

2. 另外在Python中格式化字符串什么时候执行?例如如果我的loggin的优先级设置为高,那么我还能用 `%` 操作符吗?如果是这样的话,有什么方法可以避免吗?

    log.debug("some debug info: %s" % some_info)

先回答第一个问题... `.format` 在许多方面看起来更便利.你可以重用参数,但是你用 `%` 就不行.最烦人的是 `%` 它无法同时传递一个变量和元组.你可能会想下面的代码不会有什么问题:

```
"hi there %s" % name
```

但是,如果 `name` 恰好是 `(1,2,3)`,它将会抛出一个 `TypeError` 异常.为了保证它总是正确的,你必须这样做:

```
"hi there %s" % (name,)   # 提供一个单元素的数组而不是一个参数
```

但是有点丑. `.format` 就没有这些问题.你给的第二个问题也是这样, `.format` 好看多了.

你为什么不用它?

- 不知道它(在读这个之前)
- 为了和Python2.5兼容

回答你的第二个问题,字符串格式和其他操作一样发生在它们运行的时候.Python是非懒惰语言,在函数调用前执行表达式,所以在你的 `log.debug` 例子中, `"some debug info: %s"%some_info` 将会先执行,先生成 `"some debug info: roflcopters are active"` ,然后字符串将会传递给 `log.debug()`

重复问题

rank	▲	☆	vote	url
72	349	49	512	url

怎么样去除空格(包括tab)?

有什么函数既可以去掉空格也能去掉tab?

在两侧的空白:

```
s = "  \t a string example\t  "
s = s.strip()
```

右边的空白:

```
s = s.rstrip()
```

左边的空白:

```
s = s.lstrip()
```

也可以指定字符来去除:

```
s = s.strip(' \t\n\r')
```

上面的将会去掉在 `s` 左右的空白,\t,\n和\r字符.

rank	▲	☆	vote	url
73	347	92	445	url

把文件一行行读入数组

怎么样才能一行行的读一个文件并把每行作为一个元素存入一个数组？

我想读取文件的没一行，然后把每行加入到数组的最后．我没找到这样的方法而且我也没能在Python中找到创建字符数组的方法.

```
with open(fname) as f:
    content = f.readlines()
```

我想你说的是［list］(http://docs.python.org/glossary.html#term-list)而不是数组.

rank	▲	☆	vote	url
74	345	88	440	url

和前面的type基本相同,未翻译

rank	▲	☆	vote	url
75	343	88	324	url

在Python中如何直达搜一个对象是可迭代的?

有没有一个`isiterable`的方法?我找到的解决方法:

```
hasattr(myObj, '__iter__')
```

但是我不知道这是不是最好的.

鸭子类型

```
try:
    iterator = iter(theElement)
except TypeError:
    # not iterable
else:
    # iterable

# for obj in iterator:
#     pass
```

类型检查

用抽象基类.至少Python2.6以上而且只针对新式类.

```
import collections

if isinstance(theElement, collections.Iterable):
    # iterable
else:
    # not iterable
```

rank	▲	✰	vote	url
76	340	217	397	url

用pip升级所有包

可不可以用pip一次性升级所有的Python包?

注:在官方的issue里也有这个需求.

内部还不支持这个命令,但是可以这样:

```
pip freeze --local | grep -v '^\-e' | cut -d = -f 1  | xargs pip install -U
```

rank	▲	☆	vote	url
77	340	83	533	url

给字符串填充0

有什么方法可以给字符串左边填充0,这样就可以有一个特定长度.

字符串:

```
>>> n = '4'
>>> print n.zfill(3)
>>> '004'
```

对于数字:

```
>>> n = 4
>>> print '%03d' % n
>>> 004
>>> print format(4, '03') # python >= 2.6
>>> 004
>>> print "{0:03d}".format(4)  # python >= 2.6
>>> 004
>>> print("{0:03d}".format(4))  # python 3
>>> 004
```

```
>>> t = 'test'
>>> t.rjust(10, '0')
>>> '000000test'
```

rank	▲	☆	vote	url
78	340	69	611	url

如何离开/退出/停用Python的virtualenv？

我正在用virtualenv和virtualenvwrapper.我能用命令很好的在virtualenv之间切换.

```
me@mymachine:~$ workon env1
(env1)me@mymachine:~$ workon env2
(env2)me@mymachine:~$ workon env1
(env1)me@mymachine:~$
```

但是如何退出虚拟机回到自己的环境中?到目前位置只有一个方法:

```
me@mymachine:~$
```

就是退出shell再新建一个.这有点烦人.有没有现成的命令?如果没有我怎么创建它?

通常在虚拟环境中在shell中输入:

```
$ deactivate
```

将会返回正常环境.

8| rank | ▲ | ✰ | vote | url | |:-:|:-:|:-:|:-:|:-:| | 79 | 337 | 160 | 361 | url |

Python中 `**` 和 `*` 参数有什么用?

见55

rank	▲	☆	vote	url
80	336	196	470	url

如何连接MySQL?

在python程序里如何链接MySQL数据库?

连接MYSQL需要3步

1 安装

你必须先安装MySQL驱动.和PHP不一样,Python只默认安装了SQLite的驱动.最常用的包是MySQLdb但是用 `easy_install` 安装很困难.

对于Window用户,你可以获取MySQLdb的exe.

对于Linux,可以下载 `python-mysqldb` (可以用 `sudo apt-get install python-mysqldb` 命令直接在命令行下载)

对于Mac用户,可以用Macport下载MySQLdb

2 使用

装完之后重启.这不是强制的,但是这样做可以减少问题.所以请重启.

然后就像用其他包一样:

```
#!/usr/bin/python
import MySQLdb

db = MySQLdb.connect(host="localhost", # your host, usually localhost
                     user="john", # your username
                      passwd="megajonhy", # your password
                      db="jonhydb") # name of the data base

# you must create a Cursor object. It will let
#  you execute all the queries you need
cur = db.cursor()

# Use all the SQL you like
cur.execute("SELECT * FROM YOUR_TABLE_NAME")

# print all the first cell of all the rows
for row in cur.fetchall() :
    print row[0]
```

当然,还有许多用法和选项,我只是举了一个基本的例子.你可以看看文档.A good starting point.

3 高级用法

一旦你知道它是如何工作的,你可能想用ORM来避免手动写入SQL,来把表变成Python对象.Python中最有名的ORM叫做SQLAlchemy

我强烈推荐:你的生活质量肯定能获得提高.

最近在Python里又发现了一个好东西:peewee.它是个非常轻巧的ORM,非常容易安装和使用.我的小项目和独立app都使用它,而那些工具像SQLLAlchemy或者Django用在这里有点小题大做了:

```
import peewee
from peewee import *

db = MySQLDatabase('jonhydb', user='john',passwd='megajonhy')

class Book(peewee.Model):
    author = peewee.CharField()
    title = peewee.TextField()

    class Meta:
        database = db

Book.create_table()
book = Book(author="me", title='Peewee is cool')
book.save()
for book in Book.filter(author="me"):
    print book.title

Peewee is cool
```

这个例子可以运行.除了peewee(`pip install peewee`)不需要别的的依赖.安装不复杂.它非常cool!